Uncertainty Propagation in Data Processing Systems

Ioannis Manousakis
Microsoft Corporation
iomanous@microsoft.com

Íñigo Goiri, Ricardo Bianchini
Microsoft Research
{inigog,ricardob}@microsoft.com

Sandro Rigo
University of Campinas
sandro@ic.unicamp.br

Thu D. Nguyen
Rutgers University
tdnguyen@cs.rutgers.edu

ABSTRACT

We are seeing an explosion of uncertain data—i.e., data that is more properly represented by probability distributions or estimated values with error bounds rather than exact values—from sensors in IoT, sampling-based approximate computations and machine learning algorithms. In many cases, performing computations on uncertain data as if it were exact leads to incorrect results. Unfortunately, developing applications for processing uncertain data is a major challenge from both the mathematical and performance perspectives. This paper proposes and evaluates an approach for tackling this challenge in DAG-based data processing systems. We present a framework for *uncertainty propagation* (UP) that allows developers to modify precise implementations of DAG nodes to process uncertain inputs with modest effort. We implement this framework in a system called UP-MapReduce, and use it to modify ten applications, including AI/ML, image processing and trend analysis applications to process uncertain data. Our evaluation shows that UP-MapReduce propagates uncertainties with high accuracy and, in many cases, low performance overheads. For example, a social network trend analysis application that combines data sampling with UP can reduce execution time by 2.3x when the user can tolerate a maximum relative error of 5% in the final answer. These results demonstrate that our UP framework presents a compelling approach for handling uncertain data in DAG processing.

CCS CONCEPTS

• **Computer systems organization → Architectures**;

KEYWORDS

Uncertainty Propagation, DAG Data Processing

ACM Reference Format:
Ioannis Manousakis, Íñigo Goiri, Ricardo Bianchini, Sandro Rigo, and Thu D. Nguyen. 2018. Uncertainty Propagation in Data Processing Systems. In *Proceedings of ACM Symposium on Cloud Computing, Carlsbad, CA, USA, October 11–13, 2018 (SoCC '18),* 12 pages.
https://doi.org/10.1145/3267809.3267833

SoCC '18, October 11–13, 2018, Carlsbad, CA, USA

ACM ISBN 978-1-4503-6011-1/18/10…$15.00
https://doi.org/10.1145/3267809.3267833

1 INTRODUCTION

Data is being produced and collected at a tremendous pace. The need to process this vast amount of data has led to the design and deployment of data processing systems such as MapReduce, Spark and Scope [7, 32, 37]. These frameworks typically allow data processing applications to be expressed as directed acyclic graphs (DAGs) of side-effect free computation nodes, with data flowing through the edges for processing. The frameworks then run applications on clusters of servers, transparently handling issues such as task scheduling, data movement, and fault tolerance.

At the same time, there is an urgent need for processing an exploding body of data with *uncertainties* [4]. For example, data collected using sensors are always estimates that have uncertainties—the differences between the estimated and true values—due to sensor inaccuracies. Data uncertainties also arise in many other contexts, including probabilistic modeling [10], machine learning [26], approximate storage [29], and the use of sampling-based approximation that produce estimated outputs with error bounds [2, 11].

For many applications, uncertain data should be represented as probability distributions or estimated values with error bounds rather than exact values. *Failure to properly account for this uncertainty may lead to incorrect results.* For example, Bornholt et al. have shown that computing speeds from recorded GPS positions can lead to absurd values (e.g., walking speeds above 30mph) when ignoring uncertainties in the recordings [4]. Unfortunately, developing applications for processing uncertain data is a major challenge from both the mathematical and performance perspectives. Thus, in this paper, we propose and evaluate a general framework that significantly eases this challenging task. Embedding such a framework in systems such as MapReduce and Spark will make it easily available to many developers working in many application domains.

Our framework is based on techniques that allow programmers to modify precise implementations of DAG computation nodes to handle uncertain inputs with modest effort. Uncertainties can then be propagated locally across each node of the DAG from the point where they are first introduced to the final outputs of the computation. More specifically, we use Differential Analysis (DA) [3] to propagate uncertainties through DAG nodes that are continuous and differentiable functions. For semi-continuous functions, we propagate uncertainties through a combination of DA and Monte Carlo simulation, where our framework automatically selects the appropriate method based on the input distributions and the locations of function discontinuities. For all other function types, we use Monte Carlo simulation.

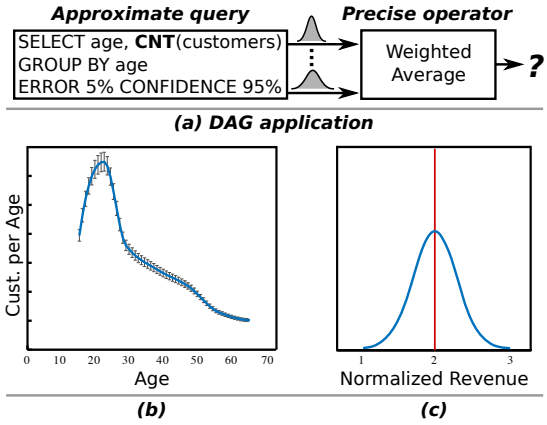

Figure 1: (a) A DAG application with two logical nodes; (b) a possible set of outputs (blue line with error bars) from the first node; (c) the output from the second node should be a probabilistic quantity rather than a precise value.

As an example of how a developer uses our framework, suppose a company needs to run a revenue prediction model implemented by a two-node logical DAG [1] shown in Figure 1(a). The first node approximates the number of customers belonging to different age groups in a database using BlinkDB [2]. The second node then computes the revenue as a weighted average, with the (uncertain) weights representing the predicted revenue per customer in a given age group. While the approximation can significantly reduce the execution time of the first node, it produces estimates with uncertainties (error bounds), rather than precise values. A developer can use our proposed framework to handle these uncertainties in the second node by providing the derivatives for the weighted average, which are essentially just the weights, with very few code changes to the precise version. This small amount of additional work will allow the answer to be computed as a distribution rather than an exact value that gives a misleading impression of precision. In particular, a precise answer, e.g., red line in Figure 1(c), may predict high revenue leading to profit while ignoring the left side of the distribution in Figure 1(c), which indicates a significant possibility of low revenue leading to an overall loss. Ignoring this possibility can be dangerous if the company is risk-averse.

We implement the proposed framework in UP-MapReduce, an extension of the Hadoop MapReduce, to handle uncertainty propagation (UP). UP-MapReduce allows programmers to develop applications with UP in much the same way as their precise counterparts. Added efforts come in the form of selecting the appropriate uncertain Mapper and Reducer classes provided by UP-MapReduce and respecting some required constraints on code structures (Section 5). Developers can optionally provide closed-form derivatives for DAG nodes that implement UP with DA to enhance performance.

We then leverage UP-MapReduce to build a toolbox of operations (e.g., sum, multiply, logarithm) on uncertain data and modify ten applications, including AI/ML, image processing, trend analysis, and model construction applications, to process uncertain data. Our experience shows that UP-MapReduce is easy to use. Running two of these applications on real data sets demonstrates the tremendous potential for combining sampling-based approximation (early in the DAG) with UP to reduce execution time while properly propagating the introduced uncertainties to the final outputs. This propagation allows users to intelligently trade off accuracy for execution time. For example, in one application, execution time can be reduced by 2.3x if the user can tolerate errors of up to 5%. Further, in one of the two applications, the original data set is a sample of network probes and so any computation on this sample necessarily has to deal with uncertainties. UP-MapReduce allows developers to easily tackle these uncertainties.

We also perform extensive sensitivity analyses on small to large execution DAGs (ranging up to tens of thousands of nodes), using eight of the applications with synthetic data, which allows us to adjust various input characteristics. Specifically, we explore the impact of UP on the magnitudes of uncertainties (e.g., whether uncertainties become worse after propagation), the accuracy of our UP techniques, overheads of UP, and scalability. Our results show that our UP techniques are highly accurate in most cases. Furthermore, the performance overheads of UP using DA are very low – average of 6% performance degradation – when closed-form derivatives are provided. Performance overheads are more significant when using DA with numerical differentiation or Monte Carlo simulation as input size increases, but this performance impact can be reduced by adding computation resources. Recall that these overheads arise from the need to process uncertain data instead of exact values. Finally, our results demonstrate that UP-MapReduce scales well to a cluster with 512 servers.

In summary, our contributions include: (1) identifying existing theories appropriate for UP and showing how to apply them to DAG-based data processing frameworks, (2) designing and implementing our proposed UP approach in a MapReduce framework called UP-MapReduce, (3) implementing a suite of data processing applications to explore the accuracy, performance, and scalability of UP-MapReduce, and (4) showing that our approach is highly effective in many scenarios, allowing applications to efficiently account for data uncertainties.

2 BACKGROUND AND RELATED WORK

In this section we first motivate the necessity for UP by presenting a (non-exhaustive) list of common uncertainty sources where UP is required if the data uncertainties are not to be ignored. We then proceed to discuss related work and in particular recent approximate methods that generate uncertainty as byproducts of the approximation. Finally, we review previous work in uncertainty estimation and belief propagation.

Sources of Uncertainty. Collecting data from imprecise instruments such as temperature, position or other analog sensors often introduces *measurement uncertainty*. In these applications, acquiring precise data is typically not an option, but it is usually possible to tune precision at the expense of resources such as more expensive sensors, higher response time or energy consumption. An example of such a trade-off is the potential for a sensor network to enter a

[1] As explained in Section 4, small logical DAGs will often map to extremely large execution DAGs with thousands of execution nodes running on large server clusters when processing large data sets.

low-power state to conserve energy at the expense of providing lower quality measurements.

Similarly, *model uncertainty* is introduced when computational models used in applications do not precisely describe physical phenomena. For example, in structure strength analysis, one may simulate the macroscopic impact of wind on a high-level model of a bridge structure, rather than modeling the forces on the individual molecules (which might be intractable).

Approximate computing is an emerging source of *approximation uncertainty*. In this setting, it may be possible for the user to trade-off precision (how much uncertainty) against execution time and/or energy consumption. Examples include iterative refinement techniques or aggregate approximation schemes (via sampling) such as BlinkDB [2] and ApproxHadoop [11]. This is a particularly interesting scenario since execution time savings achieved via approximation may be offset by the necessity for UP in subsequent nodes of a computation DAG. Other types of approximation-induced uncertainty include statistical estimators (i.e., from maximum likelihood or a posteriori estimation) and approximate storage [29].

Approximate computing with bounded errors. Extensive past work has been done in approximate computing with quality estimates by the systems, hardware and database communities. The purpose of approximate computing is to reduce the required resources (e.g., execution time and/or energy consumption) by relaxing the precision of the output but also providing estimates on the (uncertain) output quality – for example, the mean and variance of the output values.

Most prior works have focused on reducing the input set by sampling and/or dropping computation. For example, the database community has long considered the problem through approximate query processing. There, database systems sample the input data set and/or drop sub-queries to accelerate top-level queries with the ultimate goal of reducing response time, increasing throughput [1, 16, 38], and/or even providing response time guarantees [2].

Some works identify computational blocks (at compile or runtime) that can be dropped for tunable approximation with or without accuracy estimates [24, 28]. ApproxHadoop accelerates the computation of large-scale aggregations (i.e., sum, count, average) by combining sampling and computation drop [11] while providing error estimates. Others provide energy bounds by online tuning of the approximation levels [14].

Finally, the hardware community approaches the problem by trading hardware accuracy for energy efficiency, performance and transistors. For example, Esmaeilzadeh et al. [9] designed an ISA extension that provides approximate operations and proposed a micro-architecture that implements approximate functional units such as adders, multipliers, and approximate load-store units (a problem that was also tackled later by Miguel et al. [23]).

Uncertainty estimation and belief propagation. Prior work has been proposed to handle the uncertainty introduced by approximate systems and to perform belief propagation where uncertainty and prior beliefs are combined to perform inference. Approximate programming, for example, seeks to design systems and programming languages that implement and bound the errors of various arithmetic and logical operators (addition, multiplication, and comparison) when handling uncertain (probabilistic) types.

For example, Uncertain<T> [4] is a language construct that can be used to estimate the output distribution of a graph of basic operations that compose a program. Uncertain<T> can be used for inference as well, by using Bayesian statistics to derive the posterior distribution. Others have worked on probabilistic programming to implement type systems [5, 34] and compiler transformations [6, 25] to handle uncertainty, error bounding, and inference for uncertain programs. Sampson et al. [30] worked on decision making under uncertainty which is necessary to implement branches and assertions in programs. In contrast with arithmetic operations, comparison operators are more challenging, as they involve estimating the tail of the (unknown) distribution – much like our approach for UP through semi-continuous functions.

Differentiation from prior work. We differentiate from past work in uncertainty estimation as being the first to bring uncertainty propagation techniques to large-scale computational DAGs. In contrast to prior work (e.g., Uncertain <T> and ApproxHadoop) where uncertainty estimation is performed only for basic arithmetic and logical operations, we can handle arbitrary functions. At this high level of abstraction, new challenges arise. For example, accounting for covariances between uncertain data items may become a limiting factor in the performance and scalability of computations on uncertain data.

Our method also offers, to the best of our knowledge, the only known computationally tractable (and as our evaluation will show, potentially with low overheads) large-scale uncertainty propagation. Several other UP methods, such as polynomial chaos expansion and fast integration can also be used (in fact to estimate the actual distribution of $\mathbf{Y}$ instead of just computing the first two moments) [13, 19]. However, these methods are very computationally expensive, especially with increasing number of variables as noted by Lee and Chen [19]. We also do not perform any inference or multi-dimensional convolutions (as in Uncertain <T>) which suffer from high computational complexity and limit their applicability to only a few hundred input variables per node in the execution DAG. On the contrary, we show that our approach can handle millions of input variables with relatively low overhead.

3 UNCERTAINTY PROPAGATION

In this section we introduce our proposed methods for handling uncertain inputs at a DAG node. Specifically, we discuss how to (approximately) compute $\mathbf{Y} = f(\mathbf{X})$, where f is an arbitrary function without side effects, representing the computation of a DAG node, $\mathbf{X}$ is a set of random variables representing inputs with uncertainties, and $\mathbf{Y}$ is a set of random variables representing outputs with uncertainties. Depending on the nature of f (continuous, semi-continuous or discrete), we leverage a set of three statistical methods to approximate the mean μ_{Y_i} and the variance $\sigma^2_{Y_i}$ for each Y_i in $\mathbf{Y}$. These methods are described below.

3.1 UP Through Continuous Functions

We use first-order Differential Analysis (DA) to approximate the first two moments of $\mathbf{Y}$, i.e., mean and variance, for functions f that are continuous and differentiable [3]. The general strategy is to compute $\mathbf{Y}$ by approximating f using its first-order Taylor series at the expected value of $\mathbf{X}$. This approximation is accurate if f is

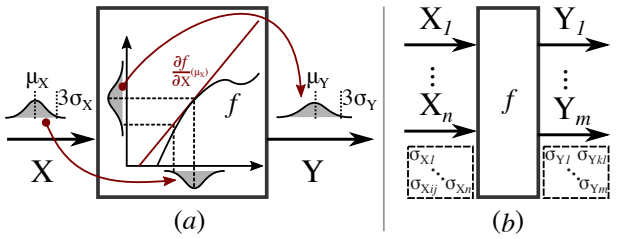

Figure 2: Uncertainty propagation through differentiable functions: (a) a single-input, single-output function, and (b) a multiple input-multiple output function.

roughly linear around the support (in other words, neighborhood) of $\mathbf{X}$; errors are being introduced otherwise. As shall be seen in Section 7.3, using the first-order Taylor series gives good accuracy for the majority of the applications we study.

For simplicity, we present DA equations for a single output value Y; we refer the reader to [3] for the full derivation of the multiple input, multiple output case. Let $Y = f(\mathbf{X})$, with $\mathbf{X} = \{X_1, X_2, ..., X_n\}$. We can compute an approximation $\hat{Y}$ of Y using the first-order Taylor series around a given point $\mathbf{X}^0 = \{X_1^0, X_2^0, ..., X_n^0\}$ as:

$$\hat{Y} = \alpha_0 + \sum_{i=1}^{n} \alpha_i (X_i - X_i^0) \tag{1}$$

$$\alpha_0 = f(\mathbf{X}^0) \text{ and } \alpha_i = \frac{\partial f}{\partial X_i}(\mathbf{X}^0)$$

We then compute an approximate mean $\hat{\mu}_Y$ by setting $\mathbf{X}^0 = \boldsymbol{\mu}_\mathbf{X} = \{\mu_{X_1}, \mu_{X_2}, ..., \mu_{X_n}\}$ and computing the expected value of $\hat{Y}$.

$$\begin{aligned} \hat{\mu}_Y = E[\hat{Y}] &= E\left[\alpha_0 + \sum_{i=1}^{n} \alpha_i (X_i - \mu_{X_i})\right] \\ &= \alpha_0 + \sum_{i=1}^{n} (\alpha_i E[X_i] - \alpha_i \mu_{X_i}) \\ &= \alpha_0 + \sum_{i=1}^{n} (\alpha_i \mu_{X_i} - \alpha_i \mu_{X_i}) = f(\boldsymbol{\mu}_\mathbf{X}) \end{aligned} \tag{2}$$

Analogously, we can derive an estimate of the variance $\hat{\sigma}_Y^2$ using the first-order Taylor series:

$$\hat{\sigma}_Y^2 = \sum_{i=1}^{n} \alpha_i^2 \sigma_{X_i}^2 + \sum_{i=1}^{n} \sum_{j=1, j\neq i}^{n} \alpha_i^2 \alpha_j^2 \sigma_{X_i X_j}^2 \tag{3}$$

where $\sigma_{X_i X_j}^2$ is the covariance of X_i and X_j. If we assume that the inputs are independent, so that $\sigma_{X_i X_j} = 0, i \neq j$, then Equation 3 reduces to the left summand.

We illustrate the computation of the mean and variance for the single-input, single-output case ($Y = f(X)$) in Figure 2(a). For the general case with multiple inputs and multiple outputs, one must also be concerned with the covariances between the outputs as shown in Figure 2(b). In general, these covariances may be non-zero. Thus, if the multiple outputs are being used as inputs to a later stage of computation as in $\mathbf{Y} = f(\mathbf{X}), \mathbf{Z} = g(\mathbf{Y})$, then we cannot assume that Y_i and $Y_j, i \neq j$, in $\mathbf{Y}$ are independent. Rather, it would be necessary to compute the covariances $\sigma_{Y_i Y_j}^2$ and use them when approximating $\mathbf{Z}$ [3].

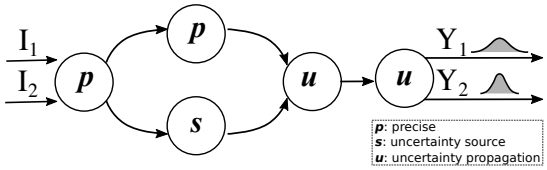

Figure 3: An example DAG where nodes labeled with p are precise computations, s introduce uncertainty (e.g., via a sampling-based approximation technique), and u require uncertain propagation.

3.2 UP Through Semi-continuous Functions

We can leverage the above approach for semi-continuous functions when the support of each X_i in $\mathbf{X}$ falls mostly or entirely within a continuous and differentiable part of the function. We adopt two approaches for checking with high confidence which intervals of $\mathbf{X}$ lie in continuous parts. The first assumes each X_i is approximately normal allowing the estimation of the support using any desired confidence interval through the corresponding covariance matrix of the input. The second approach makes no assumption about the distribution of $\mathbf{X}$. It instead uses a multivariate generalization of the *Chebyshev's inequality* [17] to bound the probability that $\mathbf{X}$ lies within any interval. For example, suppose we define a filter function as $f(X) = 1$ when $X > \alpha$ and 0 otherwise. This is a simple semi-continuous function defined on two intervals. Our framework automatically performs the required run-time checks for each X_i. In this case, it will check if X lies entirely (or mostly) in $(\alpha, +\infty)$ or $(-\infty, \alpha]$. If the condition is satisfied, it will leverage DA to propagate through the filtering function which in this case leads to an exact result. If X's support spans the discontinuity, our framework is forced to resort to Monte Carlo simulation which we discuss next.

3.3 UP Through Black-box Functions

We use Monte Carlo simulation to approximate $\mathbf{Y}$ for functions f that do not meet (or the developers to not know whether they meet) the requirements for DA. Specifically, we evaluate f on n randomly drawn samples of $\mathbf{X}$ (input) and use the outputs as an approximation of $\mathbf{Y}$. As $n \to \infty$, the empirical distribution obtained for each Y_i converges to the true distribution. To choose n, we use the following expression which bounds the difference between the empirical and the true distribution [21]:

$$P\left(\sup_{y \in \mathbf{R}} (\hat{F}_{i,n}(y) - F_i(y)) > \epsilon\right) \leq 2e^{-2n\epsilon^2} \tag{4}$$

where $\hat{F}_{i,n}(y)$ is the empirically derived CDF for Y_i and $F_i(Y)$ is the actual CDF for Y_i. For example, to approximate the CDF of $F_i(y)$ with a 99% probability of achieving an accuracy of $\epsilon = 0.05$, one would need $n = 53$ samples.

To generate accurate samples, one must know the *joint* density of $\mathbf{X}$ and pay the heavy computational cost of any rejection-sampling algorithm. Unfortunately, that cost grows exponentially with an increasing size of X and thus we resort to two approximations. The

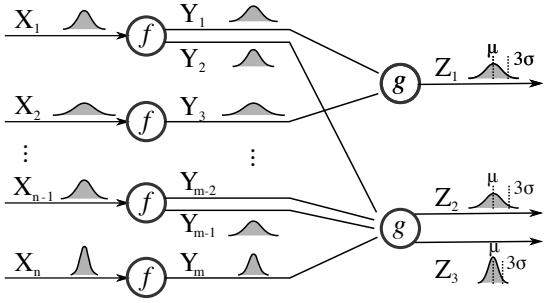

Figure 4: An example detailed view of the nodes labeled u in Figure 3, with f being computed in the first logical node and g being computed in the second logical node.

first generates samples from the input marginals X_i, when provided or previously estimated, and ignores covariances.

In the absence of full distributional information, the second approximation assumes that each input is normally distributed with the same mean and covariance matrix as the unknown distribution. Surprisingly, although the estimated distribution $\hat{Y}$ is only a coarse approximation of the actual but unknown Y, their corresponding mean and variances are similar. To see why, recall that in Eq. 3 we showed that the mean and variance estimation of Y depends solely on the mean and variance of $\mathbf{X}$. Thus, simulating (drawing samples) from any $\mathbf{X}$ that matches the required mean and variances, will accurately approximate the corresponding values for Y.

4 UP IN DAG DATA PROCESSING

We now discuss how to apply the UP techniques introduced in the last section to data processing DAGs. Figure 3 shows a small example DAG, where uncertainty is introduced in the node labeled s (e.g., via a sampling-based approximation technique). Uncertainties then must be propagated through the two u nodes following s.

Figure 4 shows an example detailed view of the two u nodes designed to highlight the challenges of implementing UP in DAG data processing. This example can correspond to transformations in a Spark program or Map and Reduce phases in a MapReduce program. This figure shows that, in general, we must handle UP through multi-input, multi-output functions for implementation in DAG data processing frameworks. Further, inputs may have non-zero covariances; e.g., Y_{m-2} and Y_{m-1} are generated from the same input, and thus are likely to have a non-zero covariance. Finally, the number of inputs and outputs may not be known statically at development time; e.g., a `reduce()` function in MapReduce has to accept an arbitrary number of values (> 0) for each key.

It is relatively straightforward to implement UP through black-box functions using Monte Carlo simulation (henceforth called UP-MC) despite the above complexities. This technique treats any node of the DAG as a black box, dynamically generates samples from the input set (each sample contains a single random value drawn from the distribution of each input data item), and dynamically computes the mean and variance for each output using the empirically derived distributions. Recall that we assume normal input distributions in the absence of this information and we ignore covariances between the inputs when constructing samples (Section 3.3), both of which may lead to inaccuracies.

The implementation of Differential Analysis (henceforth called UP-DA) is more challenging. Specifically, when a DAG node produces multiple outputs, we view it as being implemented by multiple sub-functions, each producing one of the output. For example, if a function $H(X_0, X_1, X_2)$ produces two outputs Y_1 and Y_2, then it is expressible as $Y_1 = h_1(X_0, X_1, X_2)$ and $Y_2 = h_2(X_0, X_1, X_2)$. In fact, each sub-function may depend only on a subset of the inputs; e.g., $Y_1 = h_1(X_0, X_1)$ and $Y_2 = h_2(X_0, X_2)$. In this case, the UP implementation must be able to identify the inputs used by each sub-function to correctly compute the (co) variances (Equation 3).

Thus, if a function such as f or g in Figure 4 produces multiple output values, each output must be produced by an invocation of a sub-function. The output values can be produced by multiple invocations of the same sub-function, or invocation of several different sub-functions. Each invocation must go through an UP interface so that we can track the input-to-output dependencies.

Input covariances can require additional data flow to be added to the DAG for computing output variances and covariances. For example, consider the scenario where X_1 and X_n have a non-zero covariance; even though Y_2 and Y_m are generated by different invocations of f, the covariance between X_1 and X_n will affect the variance estimates for Y_2 and Y_m. The (previously independent) computation of Y_2 and Y_m now requires the read-only covariance matrix to be present in all nodes. In general, to propagate covariances properly, each node of the DAG must have the complete covariance matrix of the sibling inputs. This requirement is challenging to implement in practice since it introduces additional data propagation and dependencies among execution DAG nodes, both of which may degrade performance and limit scalability. Our current implementation of UP in MapReduce (Section 5) does not handle all possible covariances, leaving the exploration of the full issue for future work. Meanwhile, our results in Section 7 show that this limitation does not affect accuracy significantly in most applications that we study.

As shall be seen, having closed-form partial derivatives can significantly reduce the performance overheads of UP-DA compared to numerical differentiation. Thus, an UP-DA implementation should provide an interface for the programmer to provide closed-form partial derivative functions when available. Since the number of inputs may not be known at compile time, the interface must be sufficiently flexible to allow for a parameterized implementation of the partial derivatives. For example, a function that is symmetrical on all inputs (e.g., $\sum X_i^2$) has the same partial derivative for all inputs (e.g., $2X_i$). In this case, the partial derivative can be implemented using a single function parameterized by $\mathbf{X}$ and the index i.

Finally, Figure 4 has some interesting performance implications. In the absence of covariances, UP is computed independently at each DAG node, allowing DAGs with UP to be sped up with added computation resources similar to without UP. However, speedup will ultimately be limited by the longest executing node as this "straggler" will determine the minimum execution time of the DAG. For example, g may be an aggregator function that takes inputs from many different invocations of f. If g has to aggregate a large number of inputs, then UP will require the evaluation of many partial derivatives (and possibly many numerical differentiations) for Differential Analysis or multiple evaluations of a function with many

inputs for Monte Carlo. Thus, an invocation of g with a comparatively large number of inputs can become a performance bottleneck. Fortunately, we can limit the impact of these stragglers by giving more resources to them. In particular, numerical differentiation and derivative evaluations for different inputs are independent and so can be executed in parallel. Monte Carlo runs are also independent. Parallelizing execution in both cases is quite easy, especially for many-core servers.

5 HADOOP UP-MAPREDUCE

As a proof of concept, we extend Hadoop MapReduce to include the above UP techniques in multi-stage DAG applications. We first show how our approach can be applied to the MapReduce paradigm. We then describe our implementation called UP-MapReduce.

5.1 UP-MapReduce Overview

In MapReduce, each program runs in two phases, Map and Reduce. In the Map phase, a user-written side-effect-free `map()` function is invoked per each input (key, value) pair, and produces a set of intermediate (key, value) pairs, where multiple pairs may have the same key. In the Reduce phase, a user-written side-effect-free `reduce()` function is called per intermediate key and the set of values associated with that key (produced by all invocations of `map()` during the Map phase), and produces a set of keys, each with an associated set of values [8]. MapReduce programs can further be chained together to form complex DAGs.

Figure 4 now maps readily to a MapReduce program (except that the keys are not shown), with the Map phase invoking the map function f and the Reduce phase invoking the reduce function g. It is important to note that while the MapReduce model defines that `map()` only takes one (key, value) pair as input, the value may be a set; e.g., a line of words. Thus, we implement both `map()` and `reduce()` as multi-input, multi-output functions. Assuming that only values are uncertain (keys are exact), the discussion in Section 4 applies directly to the implementation of UP-MapReduce. *Each* `map()` *or* `reduce()` *invocation expands to one or multiple UP calls through UP-MapReduce, which automatically estimates the uncertain outputs.* UP-MapReduce then streams uncertain outputs from `map()` → `reduce()` while in the case of multiple chained programs, temporarily writes these values to HDFS where the next program in the DAG consumes them.

To support functions with multiple outputs, we introduce the notions of *sub-maps* and *sub-reduces*, with each `map()` (`reduce()`) containing one or more distinct sub-maps (sub-reduces). Each output must then be produced by the invocation of a sub-map (sub-reduce) on the correct subset of inputs. We adopt similar approach for implementing semi-continuous `map()` (`reduce()`) functions (Section 3.2); user-specified continuous intervals pair with exactly one *sub-map* (or *sub-reducer* respectively).

Due to MapReduce limitations, where `map()` or `reduce()` invocations are independent, we currently do not handle the case where input covariances require additional data flow for computing output covariances; e.g., the previously mentioned case of X_1 and X_n in Figure 4 having a non-zero covariance. We do support covariances for the inputs and outputs of a single invocation.

```
class scale-map extends UPDAContinuousMapper{
  HashMap<double, double> weights;
  double k; //current key
  double eval(double contacts)
     return contacts*weights(k);
  double derivative(double in, int n, double con)
     return weights(key); //inputs are empty vectors
  void map(Text key, PV val) {
    init(key, val); //parse input and init data structures
    this.continuousUP(); //estimate mean and variance
    emit(key, new PV(this.getMean(), this.getStd()));
  }
}

class avgReducer extends UPDAContinuousReducer {
  double eval(double[] wContacts)
    return sum(wContacts)/wContacts.length;
  double derivative(double in, int n, double con)
    return 1/n; //inputs are empty vectors
  void reduce(Text key, PV vals) {
     init(key, val); //parse input and init data structures
     this.continuousUP();
    emit(key, new PV(this.getMean(), this.getStd()));
  }
}
```

Figure 5: A simple UP-MapReduce program. For readability purposes, we have changed some arrays to single objects.

5.2 Implementation

We implement UP-MapReduce as an extension of Apache Hadoop 2.7. The extension comprises three `Mapper` and three `Reducer` classes that implement UP-MC, UP-DA for continuous functions, and UP-DA for semi-continuous functions, for Map and Reduce, respectively. Developers must choose the correct classes when implementing programs for UP-MapReduce. Our extension also introduces the uncertain type `PV` (probabilistic value) which implements random variables. A `PV` variable contains one or more random variables, each described by a mean, a variance-covariance matrix, and possibly an entire empirical distribution. Below, we briefly describe the necessary `Reducer` classes. UP is implemented similarly for the `Mapper` classes.

`UPMCReducer`. This class implements UP-MC for reduce. It contains a `PV` object used to store the outputs, two abstract methods `eval()` and `reduce()`, and a `reduceWithUP()` method. The programmer needs to implement the reduce function (e.g., sum) in `eval()`, which accepts a variable number of `double` inputs and returns a variable number of `double` outputs. `reduce()` accepts a string key and a variable number of inputs in serialized form. `reduceWithUP()` implements UP-MC, and accepts a variable number of `PV` inputs. It computes the `PV` outputs using multiple invocations of `eval()` using samples derived from the `PV` inputs.

A developer would then write her Reducer class by inheriting from this class, implementing `eval()` and `reduce()`. `reduce()` should first parse the input, then call `reduceWith UP()`, and finally emit the `PV` object. It is critical that `reduce()` does *not* perform any computation on the inputs that affect the output outside of `eval()`. The developer can specify that `eval()` should be invoked multiple times, with each invocation processing a particular subset of the inputs. This feature implements the multi-input, multi-output design via sub-maps and sub-reduces.

`UPDAContinuousReducer`. This class implements UP-DA for continuous functions. The class adds an abstract method `derivative()` that accepts the inputs **X** in the form of an array of doubles, the

Application	Kernel	Shorthand
Uncertain toolbox	Basic operations	*N/A*
Matrix multiplication	Linear transformations	**mm**
Regression	Linear regression[31]	**linreg**
Clustering	k-means[15]	**kmeans**
k-nearest neighbors	p-norms [12]	**kNN**
Solving linear systems	Jacobi iteration [27]	**linsolve**
Voice/image recognition	Eigenproblem [18, 33]	**eig**
Compression	SVD [36]	**svd**
Data filtering	Filter [39]	**filter**
Trends in social media	Composite	**tsocial**
US internet latency estimation	Composite	**latency**

Table 1: Applications extended to handle uncertain inputs using UP-MapReduce.

index i to compute $\frac{\partial f}{\partial x_i}$, an array of constants that can be used as weights and outputs a double representing $\frac{\partial f}{\partial x_i}(\mathbf{X})$. The developer implements this method to provide a closed-form derivative for UP-DA. The class also implements a `reduceWith UP()` method that overrides its parent's method with an implementation of UP-DA. This method uses `derivative()` if it has been implemented, and numerical differentiation otherwise. It also uses input covariances, but expect the input covariance matrix to be loaded into the Hadoop read-only cache externally and prior to the execution of the Reduce phase. Then, it calls `eval()` as needed for evaluating the reduce function. The developer must implement `eval()` and `reduce()` as described above.

`UPDASemiContinuousReducer`. This class implements UP-DA for semi-continuous functions and inherits from `UPDAContinuousReducer`. It allows the developer to specify a list of discontinuities in the reduce function and the range of the support of each input that must be within a continuous portion of the function. It implements a `reduceWith UP()` method that checks the support of each input against the discontinuities (with the desired accuracy), and chooses to use UP-MC or UP-DA as appropriate. No further implementation is required from the developer.

Example UP-MapReduce program. Figure 5 shows the code for an UP-MapReduce program that computes a weighted average (second DAG node in Figure 1) in the presence of input uncertainties. Changes compared to a precise version are quite minimal.

Parallelization of MC and numerical differentiation. We have extended the UP Reducer implementations to use multiple threads to speed up the execution of Monte Carlo simulation and numerical differentiation on servers with multi-core and/or hyperthreaded processors.

6 APPLICATIONS

We have built a toolbox of common operations (e.g., sum) and modified ten common data processing applications using UP-MapReduce to process uncertain data. We list the applications in Table 1, along with the kernels comprising each application and shorthand names which we use later in the evaluation section. Below, we briefly discuss each one.

1) Uncertain toolbox. We apply UP-DA to a variety of continuous operations such as summation, multiplication, logarithms, exponentiation and trigonometric functions with known simple closed-form derivatives. We also include comparison and min/max operations (via UP-DA and UP-MC, respectively). We combine all the above operations to create a toolbox of uncertain elementary operations which can be used as building blocks to construct richer applications. In UP-MapReduce, these uncertain blocks may represent either a logical UP-map or a logical UP-reducer but at runtime, they will expand according to the required dataflow to one or multiple nodes in the execution DAG.

2) Matrix multiplication (mm). The multiplication of two matrices $\mathbf{A}$ ($n \times m$) and $\mathbf{B}$ ($m \times p$) can be performed by computing the elements of the output matrix $\mathbf{AB}$ ($n \times p$) as $AB_{ij} = f(\mathbf{A}_{row_i}, \mathbf{B}_{column_j}) = \sum_{k=1}^{m} A_{ik}B_{kj}$ (the inner product of $\mathbf{A}_{row_i}$ and $\mathbf{B}_{column_j}$). A MapReduce implementation can use the Map phase to read A and B and emit pairs (k_{ij}, A_{ik}) and (k_{ij}, B_{kj}) for $0 < i \leq n$, $0 < j \leq p$, and $0 < k \leq m$. The `reduce()` function can then sort the A_{ik}'s and B_{kj}'s into a sequence $A_{i,1}, A_{i,2}, ..., A_{i,m}, B_{1,j}, B_{2,j}, ..., B_{m,j}$, and then compute the inner product.

Applying UP-DA is then done as follows. The only change needed for `map()` is the handling of `PV` rather than precise values. UP is not needed because no computation is being done. The `reduce()` is rewritten to call `eval()` after properly arranging the inputs, followed by a call to `continuousUP()`. `eval()` computes the inner product. The partial derivatives for inputs from $\mathbf{A}$ is $\frac{\partial f}{\partial A_{ik}} = B_{kj}$, and vice versa for inputs from $\mathbf{B}$.

3) Regression (linreg). Fitting hyperplanes to observations is a frequent task in analytics. In particular, linear regression often relies on the least-squares method, where the sum of the squared differences between the hyperplane and the observed points is minimized. We base our application on linear regression, i.e, we are looking for $Y = \alpha X + \beta$. In the presence of noisy observations with known means and variances, we estimate the mean and variance of α and β.

4) Clustering (kmeans). Assigning observed data to clusters with k-means is frequent in data exploration. Given a fixed number of clusters and a sequence of observed data points, k-means performs an iterative algorithm, which (may) converge to a solution that minimizes the normed distance between all the points and their corresponding clusters. In the presence of uncertain data points, we extend the precise k-means algorithm with UP to estimate the mean and variance of the estimated cluster coordinates. The algorithm will then operate as a logical DAG with depth equal to the number of iterations required for k-means to converge. The logical DAG will then expand in runtime, to a large execution DAG where UP-MapReduce will propagate the uncertainty at every node. As an example, for d data points, c centroids and n iterations the uncertain execution DAG will comprise of $(d^2 \times c + 1) \times n$ nodes.

5) k-nearest neighbors (kNN). A common classification method is performed by estimating the k nearest neighbors around a data point. This computation primarily involves calculating p-norms, which measure distance in multi-dimensional spaces. We extend the traditional notion of norm with UP to estimate the mean and variance, when the input coordinates are uncertain.

6) Solving systems of linear equations (linsolve). In order to solve large ($n \times n$) systems of linear equations, in the form of $\mathbf{Ax} = \mathbf{b}$, one can use the Jacobi method to find the unknown $\mathbf{x}$. Jacobi is an iterative procedure that progressively refines the solution $\mathbf{x}$. We

extend Jacobi to support uncertain **A** and/or **b** inputs. Then, we compute the mean and variance for each element of **x**.

7) Finding eigenpairs (eig). Computing eigenpairs (and especially the dominant eigenvalue and eigenvector) is the central task in solving differential equations and computing eigenfaces. The power iteration iteratively calculates the dominant eigenpair of an input matrix. We create our own version of the power iteration to handle uncertain input data. Specifically, we combine basic uncertain operations (division), **mm** and Euclidean norms as previously shown to build the necessary iteration. The output is then a random eigenvalue and a random eigenvector.

8) Compression (svd). An effective data compression method is the Singular Value Decomposition (SVD). The SVD of an input matrix A is the key kernel in solving problems such as data compression, but also principal component analysis, weather prediction, and signal processing. We can calculate the components of SVD (U, Σ, and V) by finding the eigenvalues of AA^* and A^*A. In case A is uncertain, we extend the precise SVD implementation with UP-DA and in particular by using the uncertain toolbox and **eig**.

9) Data filter (filter). Data filters are common data manipulation tasks in large-scale data processing systems, such as Apache Spark, and built-in procedures in programming languages such as Scala. We implement an uncertain compare-aggregator filter that handles uncertain inputs. During the compare phase, the (uncertain) input data are compared against a user-defined value. The statistics of the intermediate result are forwarded to an aggregator function which estimates the uncertainty of the final result.

10) Trends in social media (tsocial). A common task in social media analysis is to study potential trends between variables of the social graph. For example, one might want to discover correlations between peoples' age and number of followers in a social media site. Assuming the data is stored in a database, a two-phase workflow (a DAG whose logical nodes execute on different DAG processing systems) will first execute a `GROUP BY` query with stratified sampling to approximate the average number of friends per age group (with each group representing one day). This stage outputs the mean number of friends and a variance for each age group. The second phase performs uncertain **linreg** between the uncertain number of friends vs. age using linear regression. It then outputs the mean and variance for the slope and intercept of the fitted line.

11) Mean US internet latency estimation (latency). Suppose that a content delivery network (CDN) operator wants to improve the average perceived latency of its customers [35]. He then seeks to maximize the US-wide 10-mile average latency by altering the position of the CDN endpoints. To perform this task, the operator first *estimates* (via sampling) the mean (10-mile) latency of some candidate locations in the US. Obviously, the operator cannot estimate the desired latency mean on every possible location in US, but instead interpolates the nearby (unobserved) locations. To correctly perform the interpolation though, one should consider that each estimated mean is actually a distribution, as every estimate is being constructed from the appropriate samples.

We replicate such a scenario and illustrate how UP can be combined in a multi-stage uncertain workflow. The workflow comprises the following stages 1) collect traceroute measurements (within the US) from the iPlanes dataset [20], 2) estimate the mean for each observed location using the samples, 3) use UP-MapReduce to perform bi-linear interpolations to estimate the mean latencies of unobserved locations and 4) use UP-MapReduce with an uncertain weighted average to simulate the frequency of packet transmission from each location based on known population density to ultimately obtain the mean and variance of the final estimate (population adjusted 10-mile mean latency).

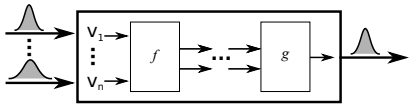

Figure 6: Monte-Carlo simulation is used to construct empirical distributions for the outputs from repeated drawing of random input samples and evaluating the *complete* application DAG for each sample.

7 EVALUATION

In this section, we evaluate UP-MapReduce by studying it's accuracy, performance, and scalability. We begin by exploring the two applications, **tsocial** and **latency**, that include sampling-based approximations and trade precision for reduced execution times. We show that by developing these applications in UP-MapReduce we can drastically decrease the execution time of both, while propagating the uncertainties introduced by the approximations. We then explore the accuracy of our UP techniques, performance overheads, and scalability via an extensive sensitivity analysis.

7.1 Evaluation Methodology

Input data sets. We leverage real datasets for the two approximate applications under study. Specifically, we evaluate **tsocial** using the Facebook social structure from SNAP social circles [22] and **latency** using traceroute measurements from iPlanes [20]. For the purpose of the sensitivity analysis (performance, precision and scalability), we generate synthetic input data sets with varying sizes and amounts of uncertainty for each application, similarly to the synthetic data generation in [40]. For each data set, we first choose a random mean value μ for each input item according to a uniform distribution on a chosen range of values. We then set the variance σ^2 for each input item to achieve a specific *relative error* defined as $3\sigma/\mu$.

Baseline. We ran a large Monte Carlo experiment that executes a precise version of each application multiple times to accurately compute the empirical distributions for the outputs. Specifically, each experiment consists of $n = 10^4$ runs of a precise application, where each run is given inputs drawn randomly according to the actual (known) input distributions. Note that this is *different* than using UP-MC for each node of an application's DAG. Here, the entire application is run from beginning to end in each run as shown in Figure 6. For an iterative application, each run executes all iterations for a given input to generate an output sample. This way, all correlations between data items passing through the DAG are correctly preserved. The output samples from the n runs are then used to construct empirical output distributions from which we extract the mean and variance for each output. We consider three different distributions for input uncertainty: normal

(Baseline-Normal), skewed with +0.5 skewness (Baseline-Skewed), and uniform (Baseline-Uniform).

Comparing UP with Baseline. We compare the mean value and relative error for each output computed by UP-MapReduce against the values produced by the corresponding Baseline experiments. When an application produces one or a small number of outputs (e.g., **linreg**), we show the comparison for the output with the largest difference between the two approaches. When an application outputs a vector or matrix (e.g., **svd**), we show the comparison using the norm of the means $\|\boldsymbol{\mu}\|_2$ and the relative error defined as $\|3\hat{\boldsymbol{\sigma}}\|_2/\|\boldsymbol{\mu}\|_2$. We expand on a case to show that using the norms do not obfuscate large differences for a subset of estimated outputs. We use the mean produced by Baseline-Normal to compute the relative error for UP in our comparisons (since the mean produced by UP is an estimate). All mean values computed by all methods were very close together, so this choice had little impact.

Experimental platform. All (but scalability) experiments were run on a cluster of 2 servers. Each server is equipped with two Intel Xeon dual-core processors, 8 GB of DRAM, 1 Gbps network interface and two 480 GB HDDs. The servers in this cluster ran Ubuntu Linux Server LTS 14.04. Scalability experiments (Section 7.4) were run on a cluster of 512 servers, where each server is equipped with two Intel Xeon 16-core processors, 64GB of DRAM, a 10Gbps network interface and four 3TB HDDs. All servers in that cluster ran Windows Server 2012. Finally, all experiments were run with UP-MapReduce (Apache Hadoop 2.7).

7.2 Approximate Computing and UP

We now leverage UP-MapReduce to build two multi-stage approximate workflows (**tsocial** and **latency**). Both first sample their initial dataset and produce uncertain intermediate values. Then, we leverage UP-MapReduce to process these uncertain values in subsequent stages, ultimately generating the final (uncertain) outputs. Our results show that UP is critical for propagating the introduced uncertainties, inform users of the magnitude of the final errors and provide guidelines to control them by adjusting the amount of initial approximation.

Specifically, **tsocial** is a two-stage approximate workflow comprising 1) the execution of an approximate query in BlinkDB [2] on 2×10^7 registered individuals, followed by 2) an uncertain linear regression (**linreg**) in UP-MapReduce. The execution of the approximate query in BlinkDB drastically reduces the execution time of the stage compared to a precise execution, but introduces uncertainties in the form of estimated errors (variance). UP-MapReduce is then used to propagate these uncertainties through the second stage of the computation. The second four-stage workflow (**latency**) approximates the mean US latency on a grid (2000 locations) by performing latency measurements only on 68 locations. This workflow comprises of 1) latency measurements which generate uncertainty due to sampling 2) generate an uncertain 2-dimensional latency surface on the obtained estimates from these 68 locations 3) perform uncertain bilinear interpolation on the (unobserved) remaining 1932 locations and 4) perform an uncertain weighted average to generate the population-weighted latency average.

Figure 7 (top) shows the execution times of **tsocial** (right y-axis) for sampling rates ranging from 0.1% to 100% (precise). It also

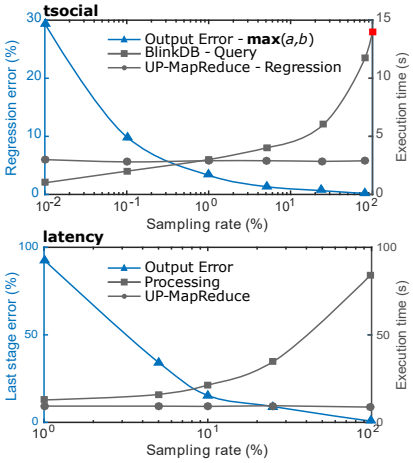

Figure 7: Obtained relative errors and execution times for varying the sampling rate of two approximate workflows (tsocial-top and latency-bottom).

shows the maximum relative error of the regression coefficients for slope and intercept (left y-axis). We only show UP-DA-numDiff for UP-MapReduce because execution times and errors for all three techniques are similar given the relatively small number of output items from BlinkDB ($\sim 3 \times 10^4$). We observe that significant savings in overall execution time can be achieved despite the overheads of UP. For example, a 5% sampling rate in the first logical DAG node leads to a relative error of just 1.35% and 51% savings in execution time (4s for BlinkDB and 2.9s for UP-MapReduce compared to 14.1s for BlinkDB without sampling plus a negligible amount of time for the precise linear regression computation). Overheads from UP require a sampling rate of 80% before approximation can lead to time savings for the workflow. After that, reduction in execution time increases as the sampling rate decreases since the UP overheads are relatively constant. Reduction in workflow execution time continues to increase until the smallest sampling rate of 0.1% for a maximum of 67.8% savings. However, the relative error increases rapidly to ~30% after a sampling rate of 1%.

Similarly, Figure 7 (bottom) shows the execution times of post-processing the obtained traceroute data (excluding the time to perform the traceroutes themselves) and the duration of the subsequent stages (UP-MapReduce bi-linear interpolation and weighed average) for sampling latencies ranging from 1% to 100%. Note that a 100% sampling rate (~2500 samples per observed location) indicates that we process all the available data; it does not correspond to sampling the entire network (which is not be possible to achieve). The estimated means are still uncertain and they include errors which should be propagated with UP.

Initially, and for sampling rates of $10 - 100\%$, we observe a generous reduction in execution time from ~82 $\rightarrow$ 19s. For smaller sampling rates, the savings cap at ~12s. The execution times for UP-MapReduce again stay unaltered as the number of observed (60) and interpolant locations (~5940) are constant (~8.1s). The output error of the weighted average, increases quadratically as we decrease sampling rate. For example, sampling just 25% of the data, we can reduce the execution time by 62.5% with an output error

of 9.01%. Similarly to **tsocial**, we only show UP-DA-numDiff, as it was the UP method with the longest running times.

Interestingly in this case, there is no trade-off between post-processing and UP-MapReduce execution times (in contrast to **tsocial**). As we always estimate the means from samples, UP-MapReduce is necessary to propagate the uncertainties. It is then evident that without UP-MapReduce, we would be unaware of the high potential workflow error (which can be as high as 92.8%).

7.3 Accuracy and Performance

We now perform a sensitivity analysis to evaluate the accuracy and performance of UP-MapReduce. We include results from all previously described applications except the toolbox, **mm** and **tsocial**, as they are included as part of the other applications under study. We start by exploring the accuracy of UP-MapReduce estimation of the means. Figure 8 plots the relative error (%) of the means (or the corresponding Euclidean norm in case of multivariate outputs) computed by UP-DA using numerical differentiation against the Baseline-Normal. These results are identical for UP-MC. We observe that UP-MapReduce estimates the means with very low bias, especially when the input relative errors are small (< 3%).

We next study the accuracy of the estimated relative errors. Figure 9(left) plots the relative errors computed by the three variants of UP-MapReduce as a function of the input relative error for 3 representative applications. The figure also plots the values produced by the three Baseline variants. Figure 9(right) plots the execution times of UP-MapReduce as a function of input size (the relative error of the input does not affect execution time). The figure also plots the execution times of precise versions, where there is zero input variance.

We observe that input uncertainties can be relatively stable, contract, or expand after propagation depending on the application. UP-MapReduce is highly accurate in most cases; i.e., its estimated relative errors are very close to the baseline values for 6 of the applications (**linreg**, **kmeans**, **latency** **filter**, **kNN** and **linsolve**). On the other hand, its estimated relative errors can also deviate noticeably from the baseline values (**eig**, and **svd**) when input errors are significant. In these cases, all three UP methods show similar deviations from the baseline although there are small differences between UP-MC and the other two approaches. Deviations for UP-DA-numDiff and UP-DA-closedForm arise from the inaccuracies introduced by Differential Analysis. Deviations for UP-MC arise from the fact that UP-MapReduce performs the Monte Carlo computation independently for each computation node in the DAG, as opposed to executing the entire DAG multiple times as in the

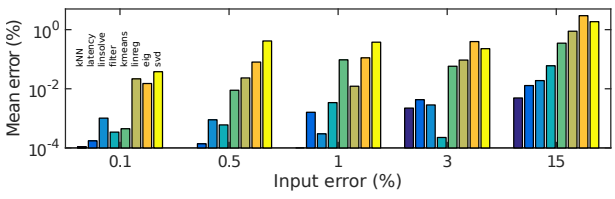

Figure 8: Relative error of means estimated by UP-DA versus Baseline-Normal.

Baseline experiments. As previously mentioned, our current implementation does not account for all covariances and does not consider input covariances when drawing input samples in UP-MC, all of which also contribute to the observed deviations.

To verify that the computed norms are not obfuscating large differences between the UP-MapReduce estimates and baseline results, we also study the differences for each output in the multi-output applications. For example, Figure 10 plots CDFs of relative errors produced by the Baseline-Normal and UP-DA-numDiff when running **linsolve** for a 50×50 linear system. Observe that UP accurately estimates the entire relative error CDF of multivariate outputs for 1% input relative errors (Figure 10(a)), while for larger relative errors of 15% UP precisely estimates a significant portion of errors (79%), with significant deviations for only a very few outputs. We observe similar trends in the remaining multivariate applications (**svd**, **kmeans**, **linreg**, **eig** and **latency**).

Interestingly, UP-DA-closedForm adds very little overhead to the precise version. This is because the derivatives for all functions being evaluated in our applications are simple functions. For example, the partial derivative with respect to x_i of the inner product $\langle \mathbf{x}, \mathbf{y} \rangle = \sum_{i=1}^{n} x_i y_i$ is simply y_i, an $O(1)$ computation. Thus, even though the number of evaluations of the derivative functions grows linearly with the number of inputs, each evaluation is extremely cheap and so the computations adds little overhead overall. Out of the eight applications, the maximum overhead (compared to the same application without UP) is 11.4% (**kmeans**) while the average across them is 6.0%.

It is important to note that the overhead for UP-DA-closedForm in general depends on the complexity of the derivatives; however, in all applications under consideration it was less expensive to evaluate derivatives of a function than the function itself. Thus, we expect the overheads of UP-DA-closedForm to be routinely lower than the ones for the other two UP techniques.

7.4 Scalability

We finally explore the scalability of UP-MapReduce by running applications 3-11 on a cluster of 512 servers. We also run the original precise applications. We choose the following input sizes: **linreg** ($16 \cdot 10^6$), **kmeans** (10^7), **kNN** ($16 \cdot 10^6$), **linsolve** ($9 \cdot 10^6$), **eig** ($9 \cdot 10^6$), **svd** ($9 \cdot 10^6$), **filter** ($16 \cdot 10^6$), **latency** ($16 \cdot 10^6$ from 150 locations). We illustrate our results (speedups) vs. increasing number of servers from four representative applications in Figure 11. The rest follow similar trends. We draw the following conclusions.

First, we observe that in all evaluated applications UP-DA-closed Form achieves similar (on average 1.6% difference) scalability as the precise version due to it's low additive overhead. Thus, uncertainty propagation does not deteriorate scalability (which as shown in Figure 11 may be poor) of the original application. Second, UP-DA-closedForm and UP-MC-monteCarlo show better scalability in all applications (except **kmeans**) due to the increased work per task (map and/or reducer) which amortizes the framework overheads. We expect this improvement to hold in applications where UP does not cause heavy execution imbalance (following observation).

Third, when evaluating **kmeans** with UP-DA-closedForm or UP-MC-monteCarlo we observe *lower* scalability (in contrary with the previous point). A study of executions shows that this is a

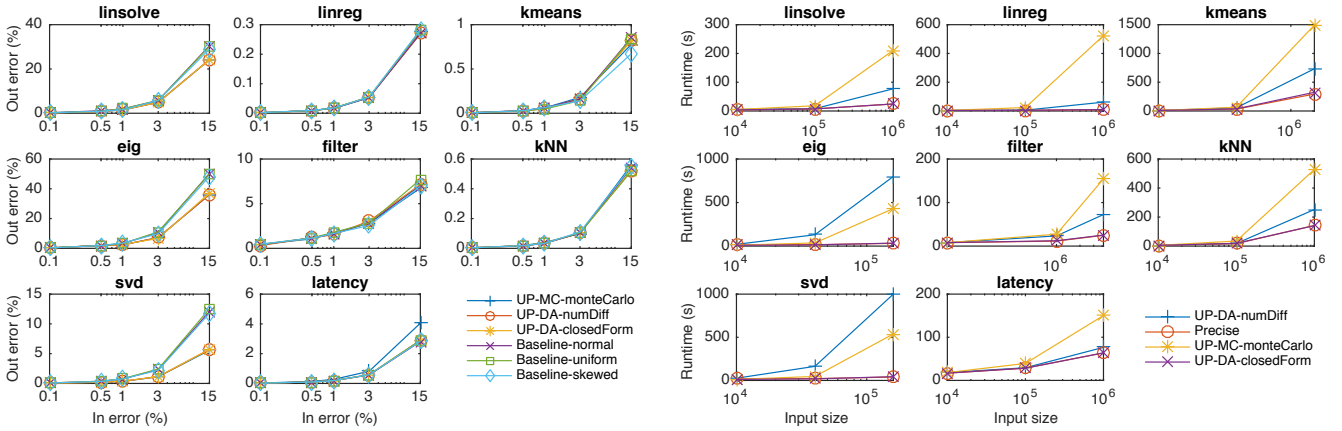

Figure 9: Precision (left) and performance (right) of UP compared to Baseline.

result of straggler reducers which are caused from data imbalance between different intermediate keys (centroids) and amplified either by the numerical differentiation or the Monte Carlo simulation. The imbalance is not noticeable in the precise and UP-DA-closedForm versions but the other UP methods increase the running times causing reduced scalability. These effects are even more noticeable on UP-DA-numDiff without straggler parallelization which attains a maximum speedup of just 10 when we utilize all our available servers (not shown in Figure 11).

8 CONCLUSIONS

In this paper, we proposed an approach for propagating data uncertainties through DAG computations. Specifically, we showed how Differential Analysis can be used to propagate uncertainties through DAG nodes implementing continuous (and semi-continuous under certain conditions) and differentiable functions. Our approach falls back to Monte Carlo simulation of nodes otherwise, but uses statistical bounds to minimize overheads while achieving a target error bounds. Our approach also allows the inter-mixing of Differential Analysis and Monte Carlo simulation for different nodes within a DAG, offering flexibility in the operations supported and minimizing performance overheads

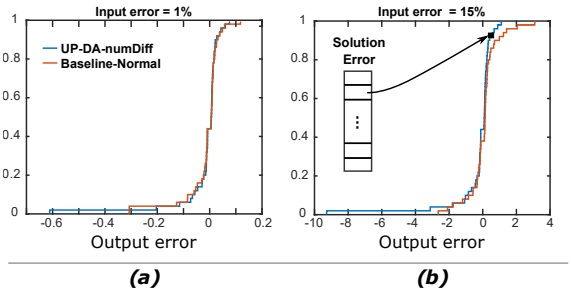

Figure 10: Comparison of multivariate error estimation (UP-DA-numDiff vs. Baseline-Normal) when solving a 50×50 linear system for input errors of 1% and 15%.

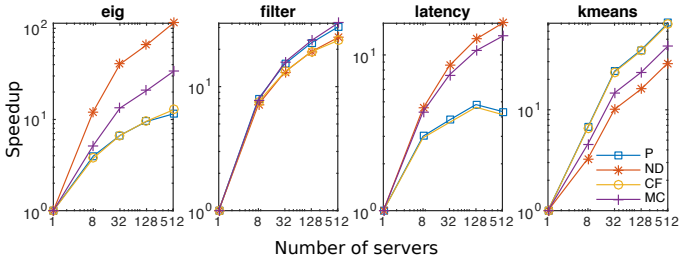

Figure 11: Scalability comparison between precise (P) and implemented UP methods (UP-MapReduce).

We have shown how our UP approach can be applied to general DAG frameworks. We have also implemented it in the UP-MapReduce system. Experimentation with ten common data analytic applications revealed that UP-MapReduce is highly accurate in many cases, while its performance overheads are very low – an average of 6% performance degradation – when closed-form derivatives are provided. When numerical differentiation or Monte Carlo simulation must be used, overheads can become much more significant as input size increases. Fortunately, the impact of these overheads on overall execution time can be reduced by allocating additional computing resources. Our scalability results show that UP-MapReduce scales well to a cluster with 512 servers. Finally, using two workflows that couple approximation with UP, we show that significant reductions in execution time can be achieved with approximation, despite the need for UP which propagates estimated uncertainties to the final output.

9 ACKNOWLEDGMENTS

This work was partially supported by NSF grant CCF-1319755.

REFERENCES

[1] Sameer Agarwal, Henry Milner, Ariel Kleiner, Ameet Talwalkar, Michael Jordan, Samuel Madden, Barzan Mozafari, and Ion Stoica. 2014. Knowing when you're wrong: building fast and reliable approximate query processing systems. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 481–492.

[2] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 29–42.

[3] Kai O Arras. 1998. *An Introduction To Error Propagation: Derivation, Meaning and Examples of Equation Cy= Fx Cx FxT*. Technical Report EPFL-ASL-TR-98-01 R3. ETH Zurich.

[4] James Bornholt, Todd Mytkowicz, and Kathryn S McKinley. 2014. Uncertain< T>: A first-order type for uncertain data. *ACM SIGPLAN Notices* 49, 4 (2014), 51–66.

[5] Brett Boston, Adrian Sampson, Dan Grossman, and Luis Ceze. 2015. Probability type inference for flexible approximate programming. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 470–487.

[6] Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C Rinard. 2012. Proving acceptability properties of relaxed nondeterministic approximate programs. *ACM SIGPLAN Notices* 47, 6 (2012), 169–180.

[7] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. 2008. SCOPE: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1265–1276.

[8] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[9] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Architecture support for disciplined approximate programming. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 301–312.

[10] Ludwig Fahrmeir and Gerhard Tutz. 2013. *Multivariate statistical modelling based on generalized linear models*. Springer Science & Business Media.

[11] Íñigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D Nguyen. 2015. ApproxHadoop: Bringing Approximations to MapReduce Frameworks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 383–397.

[12] Peter Hall, Byeong U Park, and Richard J Samworth. 2008. Choice of neighbor order in nearest-neighbor classification. *The Annals of Statistics* (2008), 2135–2152.

[13] Jon C Helton and Freddie Joe Davis. 2003. Latin hypercube sampling and the propagation of uncertainty in analyses of complex systems. *Reliability Engineering & System Safety* 81, 1 (2003), 23–69.

[14] Henry Hoffmann. 2015. JouleGuard: energy guarantees for approximate applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 198–214.

[15] Anil K Jain. 2010. Data clustering: 50 years beyond K-means. *Pattern recognition letters* 31, 8 (2010), 651–666.

[16] Shantanu Joshi and Christopher Jermaine. 2009. Sampling-based estimators for subset-based queries. *The VLDB Journal—The International Journal on Very Large Data Bases* 18, 1 (2009), 181–202.

[17] Samuel Karlin and William J Studden. 1966. *Tchebycheff systems: With applications in analysis and statistics*. Interscience New York.

[18] Patrick Kenny, Gilles Boulianne, and Pierre Dumouchel. 2005. Eigenvoice modeling with sparse training data. *Speech and Audio Processing, IEEE Transactions on* 13, 3 (2005), 345–354.

[19] Sang Hoon Lee and Wei Chen. 2009. A comparative study of uncertainty propagation methods for black-box-type problems. *Structural and Multidisciplinary Optimization* 37, 3 (2009), 239–253.

[20] Harsha V Madhyastha, Ethan Katz-Bassett, Thomas E Anderson, Arvind Krishnamurthy, and Arun Venkataramani. 2009. iPlane Nano: Path Prediction for Peer-to-Peer Applications.. In *NSDI*, Vol. 9. 137–152.

[21] Pascal Massart. 1990. The tight constant in the Dvoretzky-Kiefer-Wolfowitz inequality. *The Annals of Probability* (1990), 1269–1283.

[22] Julian Mcauley and Jure Leskovec. 2014. Discovering Social Circles in Ego Networks. *ACM Trans. Knowl. Discov. Data* 8, 1, Article 4 (Feb. 2014), 28 pages. https://doi.org/10.1145/2556612

[23] Joshua San Miguel, Mario Badr, and Natalie Enright Jerger. 2014. Load value approximation. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 127–139.

[24] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C Rinard. 2014. Chisel: Reliability-and accuracy-aware optimization of approximate computational kernels. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 309–328.

[25] Sasa Misailovic, Daniel M Roy, and Martin C Rinard. 2011. Probabilistically accurate program transformations. In *International Static Analysis Symposium*. Springer, 316–333.

[26] Kevin P Murphy. 2012. *Machine learning: a probabilistic perspective*. MIT press.

[27] Yousef Saad. 2003. *Iterative methods for sparse linear systems*. Siam.

[28] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. 2014. Paraprox: Pattern-based approximation for data parallel applications. In *ACM SIGARCH Computer Architecture News*, Vol. 42. ACM, 35–50.

[29] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. 2014. Approximate Storage in Solid-State Memories. *ACM Trans. Comput. Syst.* 32, 3, Article 9 (Sept. 2014), 23 pages. https://doi.org/10.1145/2644808

[30] Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn S McKinley, Dan Grossman, and Luis Ceze. 2014. Expressing and verifying probabilistic assertions. *ACM SIGPLAN Notices* 49, 6 (2014), 112–122.

[31] George AF Seber and Alan J Lee. 2012. *Linear regression analysis*. Vol. 936. John Wiley & Sons.

[32] Apache Spark. 2016. Apache Spark: Lightning-fast cluster computing. *URL http://spark. apache. org* (2016).

[33] M. A. Turk and A. P. Pentland. 1991. Face recognition using eigenfaces. In *Proceedings. 1991 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. 586–591. https://doi.org/10.1109/CVPR.1991.139758

[34] Steven Vajda. 2014. *Probabilistic programming*. Academic Press.

[35] Xiaofei Wang, Min Chen, Tarik Taleb, Adlen Ksentini, and Victor Leung. 2014. Cache in the air: exploiting content caching and delivery techniques for 5G systems. *IEEE Communications Magazine* 52, 2 (2014), 131–139.

[36] Jyh-Jong Wei, Chuang-Jan Chang, Nai-Kuan Chou, and Gwo-Jen Jan. 2001. ECG data compression using truncated singular value decomposition. *Information Technology in Biomedicine, IEEE Transactions on* 5, 4 (2001), 290–299.

[37] Tom White. 2012. *Hadoop: The definitive guide*. O'Reilly Media, Inc.

[38] Sai Wu, Beng Chin Ooi, and Kian-Lee Tan. 2010. Continuous sampling for online aggregation over multiple queries. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 651–662.

[39] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: cluster computing with working sets. *HotCloud* 10 (2010), 10–16.

[40] Andreas Züfle, Tobias Emrich, Klaus Arthur Schmid, Nikos Mamoulis, Arthur Zimek, and Matthias Renz. 2014. Representative clustering of uncertain data. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 243–252.