Jigsaw: Efficient Optimization Over Uncertain Enterprise Data

Oliver Kennedy*
EPFL & Cornell University
oliver.kennedy@epfl.ch

Suman Nath
Microsoft Research
sumann@microsoft.com

ABSTRACT

Probabilistic databases, in particular ones that allow users to externally define models or probability distributions – so called VG-Functions – are an ideal tool for constructing, simulating and analyzing hypothetical business scenarios. Enterprises often use such tools with parameterized models and need to explore a large parameter space in order to discover parameter values that optimize for a given goal. Parameter space is usually very large, making such exploration extremely expensive. We present Jigsaw, a probabilistic database-based simulation framework that addresses this performance problem. In Jigsaw, users define what-if style scenarios as parameterized probabilistic database queries and identify parameter values that achieve desired properties. Jigsaw uses a novel "fingerprinting" technique that efficiently identifies correlations between a query's output distribution for different parameter values. Using fingerprints, Jigsaw is able to reuse work performed for different parameter values, and obtain speedups of as much as 2 orders of magnitude for several real business scenarios.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Database Applications—*Scientific databases, Statistical databases*

General Terms

Algorithms, Design, Performance

Keywords

Probabilistic database, Monte Carlo, Black box, Simulation

1. INTRODUCTION

Enterprises often need to evaluate business scenarios to assess and manage financial, engineering, and operational risks arising from uncertain data. Our experience working with a Microsoft Windows Azure cloud platform analytics team has revealed an increasing need for tools for developing timely plans for the expansion, deployment, and allocation of resources. When making these plans, which can involve dispensation of millions of dollars, accurate and efficient simulation of many different business scenarios is critical to establish the validity of specific decisions in a timely manner. Consider an analyst who wants to forecast the risk of running out of processing capacity in a cloud cluster. For that, she needs to combine various predictive models for CPU core demands and availability. These models are inherently uncertain due to imprecise predictions of future workload, possible downtime, delays in deployment, etc. Without effective tools, simulating and evaluating business scenarios based on uncertain models can be extremely challenging.

Recently, a swath of probabilistic database (PDB) systems [3, 4, 5, 12, 13, 16, 18] have made probability distributions and models into first class citizens of the database world. PDB systems provide an ideal framework for analyzing business scenarios such as those described above. Some PDB systems, MCDB [12] and PIP [13] in particular, allow users to evaluate queries over user-provided distributions, defined as stochastic black-box sampling functions (also called variable-generation (VG) functions [12]). These systems use Monte Carlo sampling of the VG-Functions to approximate the results of queries. Support for user-provided distributions is an extremely useful feature for the analysts, who derive their baseline models using specialized, external tools such as R. Sparse, incomplete, or noisy data can transform even conceptually straightforward tasks (e.g., extracting the rate and volatility of demand growth) into daunting challenges that necessitate the use of such external tools.

A key challenge faced by PDB-based simulation systems arises when models are parameterized and the system must explore a large parameter space to optimize for a given goal. In the previous example, a CPU core availability model might accept a set of candidate purchase dates and apply them according to a model for how long it takes to bring the hardware online. The analyst can then identify purchase dates that minimize the cloud's cost of ownership, given a bound on the risk of overload. This is essentially a constrained optimization problem, albeit one where each iteration is an entire PDB query. The primary bottleneck in this context is the repeated (and potentially very costly) Monte Carlo estimation of query outputs for various parameter values, largely due to the expensive invocation of VG-Functions: (1) As the VG-Functions are black-boxes,

*Work performed while interning at Microsoft Research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'11, June 12–16, 2011, Athens, Greece.
Copyright 2011 ACM 978-1-4503-0661-4/11/06 ...$10.00.

it is not correct to assume a relationship (e.g., monotonic, continuous, etc.) between a VG-Function's parameters and outputs. Consequently, the function must be evaluated for most, if not all, possible parameter values. (2) The function may need to be evaluated over a range of steps (e.g., if it describes time series data, like a daily CPU demand model), and output at each step may be dependent on prior steps (i.e., the function describes a Markovian process, as discussed in Section 4). Therefore, with parameterization, even relatively simple scenarios taking tens of minutes, or even hours to evaluate [12] will now take hours or even days. Such a slow response is unacceptable in many practical situations where a business decision must be made quickly or if parameterized what-if scenarios are to be evaluated interactively.

In this paper, we present Jigsaw, a PDB-based simulation framework that addresses the above performance problem. Jigsaw can efficiently simulate a given business scenario by combining multiple parameterized stochastic black-box functions, possibly over multiple time steps. Jigsaw is built around a simple PDB, which performs Monte Carlo simulation over entire databases. Using this PDB functionality, Jigsaw takes parameterized queries[1] and identifies parameter values that optimize a given goal or achieve a desired property.

Jigsaw's performance improvement over existing related systems comes from the following two key observations. First, outputs of many enterprise-related stochastic functions are strongly correlated under various parameter values. For example, in our earlier example, changing the purchase date of new hardware from May to July is unlikely to affect the cluster capacity in April or August (here, purchase week and the week being estimated – the "current" week are parameters). First, identifying such correlations can help avoid exploration of significant parts of the parameter space. Second, when a simulation is Markovian (where the simulation consists of a series of steps, each depending on the simulation's output for the prior step), outputs of successive steps often remain strongly correlated. This is particularly true for many processes of interest that are built around discontinuities, with discrete events occurring at random points in time (e.g., the nondeterministic date when new hardware comes online). Identifying such *Markovian dependencies* enables the automated generation of simple non-Markovian estimators. These estimators, valid for regions of the Markov chain, allow Jigsaw to skip the corresponding portions of the simulation.

Jigsaw exploits the above two observations by using *fingerprints* of stochastic functions. The fingerprint of a stochastic black box function is a concise and easily-computable data structure that summarizes its output distribution. Thus, a fingerprint can be used to efficiently determine a function's correlation with another function, or its own instantiations under different parameter values. After such correlation has been detected, Jigsaw can avoid expensive Monte Carlo estimation (and the associated VG-Function invocations) for a target point in the parameter space by using outputs for an already-explored, correlated point. The specific fingerprinting technique we use in this paper is based loosely on random testing [11], a well known technique in software engineering: the fingerprint of a parameterized stochastic function is simply a sequence of its outputs under a fixed sequence of random inputs (i.e., seed of its pseudorandom number generator). The use of a fixed set of random seeds ensures a deterministic relationship between correlated outputs of the stochastic functions.

Note that correlation in the outputs of a stochastic function might often be obvious to a human onlooker. For example, excepting the days immediately after a hardware purchase, the day-by-day output of a simple cluster capacity model may be built out of the same distribution. However, forcing function authors to express such correlations through metadata is undesirable, as it negates the generality and clean abstraction offered by VG-functions. Data-dependent corner cases, discontinuities, and Markovian dependencies can make the metadata just as, or even more complex than the stochastic function itself. The stochastic nature of these black box functions only complicates matters further. Therefore, one important design goal for Jigsaw is to identify such correlation automatically, and fingerprints are a step towards achieving that goal.

In summary, we make the following contributions in this paper. First, we propose Jigsaw, an efficient framework for running optimization queries over uncertain enterprise data (Section 2). It uses a combination of parameter exploration, Monte Carlo simulation, and Markovian process evaluation. Second, we propose an efficient mechanism for computing fingerprints of stochastic black-box functions (Section 3). We demonstrate the use of fingerprints in the following cases: (1) We show how to use fingerprints to efficiently explore the large parameter space of an optimization query, based on a parameterized Monte Carlo simulation (Section 3). Fingerprints are used to identify similar parameter values across which VG-Function invocations may be reused. (2) We show how to use fingerprints to accelerate the evaluation of some Markov processes (Section 4). By comparing the fingerprint to that of a state-independent estimator, we can potentially avoid computing many iterations of the process. (3) We also describe how fingerprints can be used to improve the efficiency of an interactive, online scenario evaluation and parameter exploration tool called Fuzzy Prophet, which we have built on top of a PDB (Section 5). This tool uses basic probabilistic database techniques to allow users to instantiate business scenarios in the familiar environment of SQL, and analyze the effect of changing various scenario parameters with immediate feedback. Finally, we evaluate Jigsaw with several real-world scenarios seen in an enterprise cloud management team (Section 6). The results show effectiveness of our various optimizations: Jigsaw is able to improve simulation performance by as much as 2 orders of magnitude.

Note that Jigsaw has been developed in concert with an analytics team in Microsoft Windows Azure cloud platform, and hence the examples we use throughout the paper are in the context of managing a cloud infrastructure. However, we believe that the need for parameter optimization tools in PDB systems is a much more broad requirement.

2. RUNNING SIMULATIONS

In this section we first describe how Jigsaw looks like to a user via a few simple examples. We then briefly describe

[1] We assume a discrete-finite domain for each parameter. This is a reasonable expectation in our target application. Furthermore, with an infinite parameter domain, continuity must be assumed for an optimization problem to be feasible.

how Jigsaw answers queries and point out specific challenges that we address in this paper.

2.1 Probabilistic Databases

Probabilistic database (PDB) systems allow users to pose queries over uncertain data: data specified as a distribution over a range of possible values, rather than as one specific value. For example, OCR software might have difficulty distinguishing between a 9 and a 4. A PDB records both values and their corresponding probabilities. When the data is queried, the response is phrased as a distribution over possible results (although this distribution may be represented as an expectation, maximum likelihood, histogram, etc.). Several PDB implementations can similarly represent and query continuous distributions (e.g. a gaussian distribution representing a measurement and its statistical error).

While a traditional DBMS stores a single instance of a database, a database in a PDB system represents a distribution over a (potentially infinite) set of database instances typically referred to as possible worlds. Queries in a PDB are (conceptually) evaluated by evaluating the query in each possible world. The set of results forms an answer distribution for the query. Clearly, this approach is not feasible in general, so some PDBs compute approximate results using Monte Carlo methods. The MCDB [12] system, on which Jigsaw's PDB implementation is based, instantiates a finite set of databases by sampling randomly from the set of possible worlds. Queries are run on each sampled world in parallel, and the results are aggregated into a metric (e.g. an expectation or standard deviation of the result) or binned into a histogram.

Note that MCDB's interaction with distributions being queried is limited to the generation of samples. This simple interface makes it possible for users to incorporate (nearly) any distribution into their queries. User-defined probability distributions (e.g. a user demand forecast) can be incorporated by constructing a stochastic black-box function referred to as a VG-Function, which generates samples drawn from the distribution.

2.2 Example queries in Jigsaw

Batch mode execution. Suppose that, in our context of cloud infrastructure, an analyst wishes to determine the optimal date and volume for several server purchase orders to keep the risk of running out of available CPU cores below a certain threshold. The later the purchases occur, the lower the hardware's upkeep costs, but the greater the chance that cores will be unavailable when needed. The question of an ideal purchase date and volume is a simple constrained optimization problem.

A Jigsaw user would specify this optimization problem in three stages: (1) The user defines stochastic models forecasting CPU core availability and demand, (2) The user specifies inter-model interactions to describe the scenario, and (3) Jigsaw solves the optimization problem by exploring the parameter space of purchase dates and volumes.

For step (1), the user defines various stochastic models that Jigsaw uses as black boxes. These stochastic black boxes are essentially functions that produce samples[2] drawn

```
-- DEFINITION --
DECLARE PARAMETER @current_week AS RANGE 0 TO 52 STEP BY 1;
DECLARE PARAMETER @purchase1 AS RANGE 0 TO 52 STEP BY 4;
DECLARE PARAMETER @purchase2 AS RANGE 0 TO 52 STEP BY 4;
DECLARE PARAMETER @feature_release AS SET (12,36,44);

SELECT DemandModel(@current_week, @feature_release)
          AS demand,
       CapacityModel(@current_week, @purchase1, @purchase2)
          AS capacity,
       CASE WHEN capacity < demand THEN 1 ELSE 0 END
          AS overload
  INTO results;
-- BATCH MODE --
OPTIMIZE SELECT @feature_release, @purchase1, @purchase2
         FROM results
         WHERE MAX(EXPECT overload) < 0.01
         GROUP BY feature_release, purchase1, purchase2
      FOR MAX @purchase1, MAX @purchase2
```

Figure 1: An example Jigsaw query.

from the probability distribution that they intend to describe. This framework allows analysts to easily import externally defined models that describe a wide variety of processes and system characteristics. In our specific example, the user writes the following two functions (e.g. based on a model derived in a statistical modeling application like R):

```
DemandModel(current_week, feature_release);
CapacityModel(current_week, purchase1, purchase2);
```

The `DemandModel` function produces a stochastic CPU core usage demand forecast for a given week in the future, taking into account expected future user arrival rates, individual user capacity requirements, and expected user reactions to planned special offers and system features.

The `CapacityModel` function outputs a stochastic estimation of the number of CPU cores available on a given date in the future (given a set of future purchase dates). It also takes into account the current CPU core availability, future expected failure rates, and prediction, based on prior purchasing experiences of when new cores will come online after purchase. How the models are developed is orthogonal to Jigsaw's execution engine, which simply treats the models as black boxes. This way, we establish a clean separation between expert knowledge and the task of simulating the underlying process.

For steps (2) and (3), the user writes the SQL-like query in Figure 1. The core of the scenario is a simple SQL `SELECT` query that produces an output `result` table – in this example, containing `capacity`, `demand`, and `overload` columns. Note that, as Jigsaw is built around a PDB system, this results table is specified as a probability distribution over the space of possible results. Two aspects of the query require further discussion: (a) The query contains several parameter variables, each prefixed with a `@`. Parameter variables, with their bounds and sets of permitted and initial values, are declared as part of the scenario using `DECLARE PARAMETER` statements and are equivalent to standard SQL variables from the user's perspective. (b) The optimization goal is expressed with an `OPTIMIZE` query, which iterates over the parameter space to find the latest `purchase1` and `purchase2` that keep the expected risk of `overload` (a condition defined as a week when `capacity < demand`) within a threshold.

[2]Canonical VG-Functions in MCDB produce tables as output. For clarity, this paper uses a simplified notion of stochastic black-box functions that produce only single values. We make this distinction explicit by using the term black-box function. Naive extensions of Jigsaw's fingerprinting technique to VG-functions are trivial (e.g., extend the function with row and column id parameters) and optimized extensions are beyond the scope of this paper.

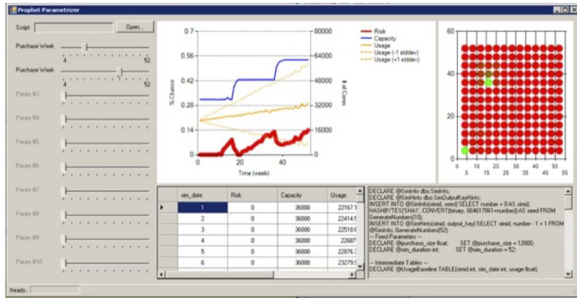

Figure 2: Jigsaw's interactive interface.

One possible implementation of the `CapacityModel` is of interest. This model is charaterized by a sequence of discrete events (e.g., purchases or hardware failures), each affecting the cluster's capacity. Each event is produced by a separate model, so the database engine itself can compute the cumulative effect of the events with a simple SQL `SUM` aggregate. Also consider the `CapacityModel` expectation viewed in a time-series plot. Though each purchase has a stable long-term impact on the cluster's capacity, this plot is characterized by two distinct *structures* in the vicinity of each purchase date. We will return to this observation later.

Interactive mode. Jigsaw can also be used in an interactive online mode. In this mode, the user modifies various parameter values (e.g., purchase date and volume) and quickly sees the outcome (e.g., the risk of overload at a given date). As parameter values are modified, the system continually updates a progressively refined estimate of the results table for those parameter values. This quickly gives a rough estimate of the final answer so that the user, not finding the given parameter values interesting, can abandon the simulation in the middle and try a different parameter value. This mode is particularly targeted at users who may not have an extensive statistics background. An analyst-developed scenario can be used by an executive (e.g. as part of a management dashboard tool) to quickly observe the expected outcome of specific financial decisions for various parameter values.

The interactive mode, with the output shown in Figure 2, is expressed with the following execution query (parameter definition and `SELECT` portions of the query are same as in Figure 1):

```
-- INTERACTIVE MODE --
GRAPH OVER @current_week
      EXPECT overload WITH bold red,
      EXPECT capacity WITH blue y2,
      EXPECT_STDDEV demand WITH orange y2;
```

The query above provides Jigsaw with a parameter to use as the graph's X-Axis, and specifies how each column in the results table is to be graphed in the GUI (Figure 2).

2.3 Jigsaw Simulation Process

Figure 3 shows how a Jigsaw executes an optimization query in the batch mode. Each random table in the uncertain database is represented on disk by its schema, together with a set of black-box functions that are used to generate realizations of uncertain attribute values. When a query is issued, the *Parameter Enumerator* module enumerates all feasible parameter values for the black-box functions involved in the query. This brute force approach is necessary to guarantee that the optimization converges to the global maximum for an arbitrary black-box function. Note that Jigsaw's fingerprinting techniques remain applicable to more advanced techniques that use additional information about the black-box (e.g., gradient-descent, if the black-box is known to be continuous).

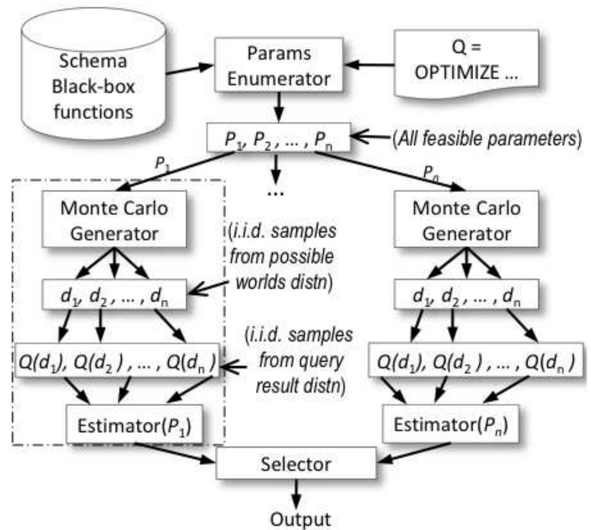

Figure 3: Processing optimization queries in Jigsaw

For each parameter value, Jigsaw then invokes its PDB subsystem (shown inside the dotted box). The PDB subsystem (loosely modeled after MCDB) invokes the black-box functions with the current set of parameter values to generate a set of $n \geq 1$ independent and identically distributed (i.i.d.) sampled *instances*, sometimes referred to as possible worlds; for parameter valuation P_a, we say that sample d_i is generated by instance (P_a, i). Recall that in a PDB, the output of a query is a probability distribution. Evaluating the query over each sampled possible world generates a set of i.i.d. samples of the results table's distribution. These latter samples are then aggregated by the *Estimator* to compute one or more characteristics of interest (i.e., mean, standard deviation, etc...) for the output distribution. The process is repeated for all different parameter values. Finally, the *Selector* component selects the parameter value, along with its output distribution, that satisfies the optimization goal.

2.4 Jigsaw Challenges

The most expensive aspect of Jigsaw's simulation process is its interaction with the underlying PDB. This processing overhead is linear in the size of the parameter space and dominates all other processing tasks performed by Jigsaw. The primary goal of Jigsaw is to reduce the number of instances on which the PDB must be invoked. To achieve this, we exploit several observations about redundancy in computations. First, outputs of many enterprise-related stochastic functions are strongly correlated under various parameter values (examples in Section 3). Identifying such correlations can help to avoid exploring large regions of the parameter space. Second, in many event-based processes with markovian dependencies (i.e. each step in the process depends on the output of the prior step), the markovian dependencies are relevant only in the steps near an event. A suitably crafted non-markovian estimator function (examples in Section 4) may be used to reduce simulation required for the other steps. Finally, in interactive mode execution, a

quick estimation of simulation results for a selected parameter value can often be given based on results from previously selected parameters; the estimation can then be gradually refined with more samples.

To exploit the above observations, Jigsaw needs to address the following challenges.

- How can parameter values that produce the same (or similar) outputs be efficiently identified and exploited?
- How can correlated Markovian steps be efficiently identified and exploited?
- Can an accurate estimate be obtained for one parameter value in interactive mode by reusing results computed for other parameter values?

In the rest of the paper, we discuss how Jigsaw addresses these challenges.

3. FINGERPRINTS

The key concept Jigsaw uses to reduce the number of Monte Carlo evaluations is *fingerprints*. A fingerprint of a stochastic black box function is a concise and easily-computable data structure that summarizes its output distribution. Thus, a fingerprint can be used to determine a function's *similarity* with another function, or its own instantiations under different parameter values. We will show a concrete example of fingerprints in Section 3.1, but first we explain the general principle.

The outputs of a deterministic function F evaluated on two different values P_i and P_j, are deemed similar (denoted as $F(P_i) \sim_{\mathcal{M}} F(P_j)$) if there exists a closed form mapping function $\mathcal{M}$ that maps from $F(P_i)$ to $F(P_j)$.

$$F(P_i) \sim_{\mathcal{M}} F(P_j) \equiv F(P_i) = \mathcal{M}(F(P_j))$$

Consider a stochastic function F with output $X = F(P_i)$ and probability distribution $f(x = X|P_i)$. F is similar at P_i and P_j if a closed form mapping function exists to map the domain of $f(x|P_i)$ into that of $f(x|P_j)$.

$$F(P_i) \sim_{\mathcal{M}} F(P_j) \equiv \forall x : f(x|P_i) = f(\mathcal{M}(x)|P_j)$$

More generally, we can think of $\mathcal{M}$ as the central element of a family of mapping functions that map not only function values but also metrics, aggregates, and other derived values. Efficient mapping between members of this family can substantially reduce the sampling requirements of a computation. For example, consider a scenario where both expectations $E[F(P_i)]$ and $E[F(P_j)]$ are needed, and $F(P_i) \sim_{\mathcal{M}} F(P_j)$ can be efficiently proven. An $\mathcal{M}_{expect}$ derived from $\mathcal{M}$ such that $E[F(P_j)] = \mathcal{M}_{expect}(E[F(P_i)])$, eliminates the need to explicitly compute $E[F(P_j)]$.

Identifying the mapping function for an arbitrary pair of stochastic black-box functions $(F(P_i), F(P_j))$ is difficult for two reasons: (1) The functions are black-boxes – interactions with the function are limited to sample generation. (2) The functions are stochastic. In order to match two distributions, it is first necessary to approximate the distributions (i.e., by sampling from both, negating the benefits of having established similarity).

Rather than attempt to map the result distribution, Jigsaw employs a shortcut. We propose the abstract `fingerprint` operation (and corresponding mapping function M_f), which

Algorithm 1 DemandModel(*current_week*, *feature*)

Require: The *current_week* being simulated, and a *feature* release date.
Ensure: The *demand* for the week being simulated.

```
1: demand = Normal(
       μ : 1 * current_week,
       σ² : 0.1 * current_week
   )
2: if current_week > feature then
3:     demand += Normal(
           μ : 0.2 * (current_week − feature),
           σ² : 0.2 * (current_week − feature)
       )
```

efficiently maps parameterized stochastic black-box functions to concise, comparable data structures such that with high probability:

$$F(P_i) \sim_{\mathcal{M}} F(P_j) \equiv \texttt{fingerprint}(F(P_i)) = \mathcal{M}_f(\texttt{fingerprint}(F(P_j)))$$

Fingerprints can be computed for individual stochastic black-box functions, such as `DemandModel` in Figure 1, or combinations of such functions. Taken to one extreme, the entire Monte Carlo simulation shown inside the dashed box in Figure 3 can be treated as the stochastic function F. Thus, $F(P_i) \sim_{\mathcal{M}} F(P_j)$ implies that we can avoid expensive Monte Carlo simulations for parameter value P_j and estimate the output of $Estimator(P_j)$ accurately as $\mathcal{M}_{est}(Estimator(P_i))$.

3.1 Computing Fingerprints

Identifying similarities between the outputs of two functions is, in general, hard [15]. Jigsaw uses a probabilistic approach based on the principle of *random testing* [11], a well-known software testing technique. For random testing of a deterministic function F against a hypothesis function H, both functions are evaluated on m random inputs and the results are compared. The function F is declared satisfying the hypothesis H if the outputs of F and H match for all m random inputs. Random testing has two features in particular, that make it well suited to our needs: (1) Random testing is simple and can be used while treating the functions as black boxes. (2) Random testing has been shown to be robust for functions with a small number of conditional branches so that a small number of random inputs can exercise all code paths. We have found in our cloud infrastructure management context that almost all our stochastic functions are relatively simple and contain at most one or two conditional branches.[3]

We use the same principle to determine similarities of outputs of a stochastic black-box function F under two valuations of the same parameters P_i and P_j. However, unlike random testing where the parameters are random and the function is deterministic, Jigsaw must deal with stochastic functions and fixed parameters. To make F deterministic, we extend F with a seed parameter σ and ensure that all sources of randomness within $F(P_i, \sigma)$ are replaced by invocations of a pseudorandom generator seeded by σ. In

[3]The functions are kept simple in practice, modeling only one particular aspect of the system so that they can be trained and validated even with small, noisy data sets.

practice, these modifications are negligible, as randomness is typically obtained from system API calls (e.g. `rand()`).

It is crucial for both invocations of F to use *the same source of randomness* to make their comparison meaningful. Consider two stochastic functions that output 0 and 1 with equal probability. When repeatedly evaluated with the same sequence of random seeds, they can be quickly declared to be equivalent with a very high probability. On the other hand, using different seeds, equivalence testing is much more difficult. Consider our example stochastic function in Algorithm 3.1. As a sum of two normal distributions, the function's output is normally distributed for all inputs. Suppose the function is invoked twice as `DemandModel`(1,3) and `DemandModel`(2,4). Both invocations take the same code path, and their outputs will be drawn from linearly correlated distributions. In addition, by using a pseudorandom number generator seeded with the same value for each invocation, we ensure that there is not just a correlation, but a linear mapping from one fingerprint to the other. In contrast, using different random seeds would hide the one-to-one similarity in their outputs.

A Concrete Fingerprint. The fingerprint of a parameterized stochastic function $F(P_i)$, with respect to a vector of m seed values $\{\sigma_k\}$, is the vector of size m where the k'th entry of the vector is the output of $F(P_i)$ with σ_k as the random seed. More formally,

$$\texttt{fingerprint}(\{\sigma_k\}, F(P_i)) = \{\theta_k = F(P_i, \sigma_k) | 0 \leq k < m\}$$

For the remainder of the paper, we use this definition of fingerprints and an implicit global seed set $\{\sigma_k\}$, randomly generated as part of the initialization process and held constant throughout.

Note that using the same set of random seeds for different parameter values does not affect the correctness of Jigsaw's Monte Carlo simulations. Referring to Figure 3, since the seeds used by each Monte Carlo Generator are i.i.d. random, inputs to the $Estimator(P_i)$ are i.i.d. samples from query result distribution. Thus, the output of $Estimator(P_i)$ remains statistically correct. Using same set of seeds for different parameter values introduces correlated error terms into the outputs of different Estimators, but the Selector only compares, and never combines, the Estimator's outputs.

Mapping Functions. For fingerprints, as defined above, we can define fingerprint mapping functions $\mathcal{M}_f$ that can be applied to each element of a fingerprint (in order to identify its similarity with another fingerprint). For example, consider two fingerprints: $\theta_1 = (0, 1.2, 2.3, 1.3, 1.5)$ and $\theta_2 = (0.1, 1.3, 2.4, 1.4, 1.6)$. The mapping function $\mathcal{M}(x) = x+0.1$ maps θ_1 to θ_2. In general, mapping functions should be: (1) easy to parameterize, (2) easy to validate, (3) easy to compute, (4) easily applied to simple aggregate properties (e.g., expectation).

Given two fingerprints, Jigsaw can automatically compute a linear function (i.e., $\mathcal{M}_{\alpha,\beta}(x) = \alpha x + \beta$) that maps one fingerprint to another, if such a mapping exists (Algorithm 2). Linear mapping functions fulfill our desired characteristics precisely: (1) The mapping function can be determined from two distinct values in a pair of fingerprints. (2) The remaining values in the fingerprints can be used to validate the mapping. (3) Linear functions are incredibly simple, and (4) can be easily applied to simple aggregate properties such as expectation and standard deviation.

In general, the notion of similarity between two signatures is application dependent. Therefore, Jigsaw allows users to provide their own classes of mapping functions.

Algorithm 2 $FindLinearMapping(\theta_1, \theta_2)$

Require: Two fingerprints θ_1 and θ_2 of size m
Ensure: A linear function $\mathcal{M}(x) = \alpha x + \beta$ such that $\mathcal{M}(\theta_1[i]) = \theta_2[i], \forall i$, and *null* if no such function exists
1: $\alpha \leftarrow (\theta_2[1] - \theta_2[2])/(\theta_1[1] - \theta_1[2])$
2: $\beta \leftarrow \theta_2[1] - \alpha\theta_1[1]$
3: $match \leftarrow$ **true**
4: **for** $i = 3$ to m **do**
5:     **if** $\alpha\theta_1[i] + \beta \neq \theta_2[i]$ **then**
6:         $match \leftarrow$ **false**
7: **return** ($\mathcal{M}(x) = \alpha x + \beta$) **if** $match$, $null$ otherwise

Using Fingerprints. With fingerprints, Jigsaw executes Monte Carlo simulations for different parameter values as follows. Let F denote the entire Monte Carlo simulation with a parameter value P_i (i.e., the computation inside the dashed box in Figure 3). Thus, the fingerprint of $F(P_i)$ is essentially the outputs of first m simulation rounds with parameter P_i.

During execution, Jigsaw incrementally maintains a set of *basis distributions*. Each basis distribution is a tuple (θ_i, o_i), implying that Jigsaw has already computed the output metrics o_i for some $F(P_i)$ with fingerprint θ_i. For a new parameter value P_j, Jigsaw first computes fingerprint θ_j of $F(P_j)$ (as part of the first m rounds of simulation with parameter P_j). It then checks for a basis distribution with fingerprint θ_k such that $\theta_j \sim_{\mathcal{M}} \theta_k$. If such a basis distribution exists, Jigsaw omits the subsequent rounds of simulation for P_j and returns $\mathcal{M}_{est}(o_k)$ instead.

Retrieving Mapping Functions. When presented with an unknown distribution, Jigsaw compares each new fingerprint against all the fingerprints in the basis distribution, identifying a mapping to one of them if it exists. Algorithm 3 shows the process. Jigsaw first uses a suitable indexing scheme (described next) to prune the search space of candidate basis fingerprints. For each pairing candidate, Jigsaw uses the `FindMapping` function to discover a possible mapping between the two fingerprints. An instance of the `FindMapping` function, the `FindLinearMapping` function shown in Algorithm 2 searches for mappings of the form $\mathcal{M}(x) = \alpha x + \beta$. If a mapping exists between two fingerprints, Jigsaw uses the mapping to reuse work done for the existing basis distribution. If no mappable fingerprint can be found, Jigsaw completes the simulation process and adds the results (i.e., the fingerprint and computed metric(s)) to the set of basis distributions.

3.2 Indexing Fingerprints

The existence of $\mathcal{M}$ can be computed quickly for any pair of fingerprints. However, the expected number of times this test must be performed grows linearly with the number of basis distributions.

Instead of naively scanning every basis distribution, Jigsaw builds an index over the basis fingerprints. The goal of indexing is to quickly find a set of candidate basis fingerprints that are similar to a given fingerprint (i.e., where a mapping exists). The set of fingerprints returned by the index must contain all similar fingerprints. In addition, it may contain few fingerprints that are not similar to the given

Algorithm 3 $FindMatch(F, P_a)$

Require: A stochastic black box function F, and a point in its parameter space P_a.

Ensure: The pair $(basis, \mathcal{M})$, where $basis$ is a basis distribution (fingerprint θ, output metrics o), and $\mathcal{M}$ is a mapping function such that $\theta \sim_{\mathcal{M}} \texttt{fingerprint}(F(P_a))$

1: $\theta \leftarrow \{F(P_a, \sigma_i) | i \in [0, m)\}$
2: $candidates \leftarrow CandidateFingerprint(basis, \theta)$
3: **for all** $basis \in candidates$ **do**
4: $\quad \mathcal{M} \leftarrow FindMapping(basis, \theta)$
5: $\quad$ **if** $\mathcal{M} \neq null$ **then**
6: $\quad\quad$ **return** $(basis, \mathcal{M})$
7: **return** $\{[(\theta, \texttt{Estimator}(F(P_a))), (\mathcal{M}(x) = x)]\}$

fingerprint; these false positives are later discarded in Algorithm 3.

Currently Jigsaw supports the following two indexing strategies that reduce the cost of matching linear mapped fingerprints to a single hash-table lookup with high probability.

Normalization. The first indexing strategy is to translate the fingerprints to their normal forms so that two *similar* fingerprints have the *same* normal form (and hence can be retrieved by a hash lookup). Such normalization requires a class of mapping function that admits a normal form translation. For example, when using a linear mapping function, a fingerprint's normal form can be produced by taking the first two distinct sample values and identifying the linear translation that maps them to 0 and 1 (or, any two predefined constants) respectively. If two fingerprints have a linear mapping, then *all*, not just the first two, entries of their normal forms will be identical.

Sorted SID. Normalization requires that the mapping function admits a normalized representation of a fingerprint. In some cases (e.g., a probabilistic mapping), no such normal form can be computed easily. In such cases, we assign each sample value in a fingerprint an identifier (e.g., its index position in the fingerprint), using the same identifier ordering across all fingerprints. We then sort the sample values in a fingerprint, and take the resulting sequence of sample identifiers (or, SIDs) as the hash key in the index. As long as the mapping function is monotonically increasing, the resultant ordering of SIDs will be consistent across all mappable distributions. Even if the mapping function is only monotonic, a similar effect can be achieved by comparing both the SID sequence and its inverse.

4. MARKOVIAN JUMPS

Jigsaw allows users to specify inter-model dependencies. Consider two models where the first model predicts the release date of a particular feature of the cloud service, and the second model predicts demand, given that release date. Frequently, such dependencies are cyclical: the feature release date might be driven by demand. For example, sufficiently high demand might convince management to allocate additional development resources to the feature.

As a consequence of this sort of cyclical dependency, the models and thus the simulation must be evaluated as a Markovian process, where a model is evaluated in discrete steps and its output for any given step is dependent on the prior step's output. The discrete steps are usually small (e.g., a day in the above example) so that outputs of other models affecting the model remain static within a step. Every step in the process must be simulated, even if the only output of interest is for one specific step (e.g., user demand in two months).

In the space of cloud logistics, models with this sort of cyclical dependency often have an interesting characteristic: the Markovian dependency is present only over certain steps. In the case of the feature release date, as long as the user demand remains strictly (or at least with high probability) below or above the threshold value, the feature release date is unaffected. For these periods, the demand and feature release date model can be treated as non-Markovian, despite its cyclical dependency. Concretely, Markovian dependencies in this sort of model are characterized as (1) infrequent, and (2) often closely correlated (3) discontinuities in (4) an otherwise non-Markovian process. Thus, given the state of the system at the beginning of one of these non-Markovian regions, it is possible to create a non-Markovian *estimator function* for the remainder of the region.

These infrequent Markovian dependencies occur often in event-based simulations. Forcing programmers to identify the ranges within which these dependencies occur is undesirable. Instead, Jigsaw can automatically identify non-Markovian regions in these processes automatically by using fingerprints.

4.1 Fingerprinting Markov Processes

Consider a model F that needs to be evaluated in a sequence (or a chain) of discrete steps. Assuming that Markovian dependencies are infrequent, outputs of F in many successive steps will not be affected by previous steps. To *jump* over such non-Markovian steps and avoid expensive computation, Jigsaw uses a non-Markovian estimator function E (discussed further in Section 4.2), which predicts the outputs of F at different steps of the chain *without* considering the outputs (of F or other models) at previous steps. By comparing the fingerprints of E and F, Jigsaw can efficiently identify the regions over which E is a valid approximation.

Recall that each fingerprint of F is a set of its random outputs. Thus, the fingerprint for any step in a Markov process can be used to generate the fingerprint for the next step. Instead of evaluating the full set of n Monte Carlo simulation rounds of the Markov chain, we evaluate only a fingerprint-sized ($m < n$) set and compare it to the fingerprint of an estimator function. If a mapping exists between the two, the estimator remains viable.

To compute the value of a Markovian black-box function at a particular step in the chain, Jigsaw does an exponential-skip-length search of the chain until it finds a point where the estimator fails to provide a mappable fingerprint. From that point, it does a binary search to find the last point in the chain where the estimator provides a mappable fingerprint, uses the estimator to rebuild the state of the Markov process, generates the next step, and repeats the process. This algorithm is shown explicitly in Algorithm 4.

Consider the previous example of a cyclically dependent user demand and feature release date models. An example execution of the Markov Jump algorithm is illustrated in Figure 4. Jigsaw begins with an estimator for the Markov process that assumes the feature has not yet been released (the initial system state). (4.a) It iterates over each step of the Markov process, computing only the fingerprint and not

Algorithm 4 $MarkovJump(F_{mkv}, initial, target)$

Require: A function, $F_{mkv}(prev_state) = new_state$ describing a Markov process and its estimator, respectively. An *initial* state for the functions. A *target* number of steps to return after. A statically defined fingerprint size m.

Ensure: The state of each instance of the Markov process after *target* steps.

1: $state \leftarrow \{initial, initial, \ldots\}$; $\theta_1 \leftarrow state[0 \ldots m]$
2: $s \leftarrow 1$; $F_{est} \leftarrow F_{mkv}(\theta_1)$
3: **loop**
4: **for** $s/2 < i \leq s$ **do**
5: $\theta_i \leftarrow F_{mkv}(\theta_{i-1})$
6: **if** $(s > target) \wedge (F_{est} \sim_{\mathcal{M}} \theta_{target})$ **then**
7: **return** $\mathcal{M}(F_{est}(state))$
8: **if** $F_{est}(s, state[0 \ldots m]) \sim_{\mathcal{M}} \theta_s$ **then**
9: $s \leftarrow s * 2$
10: **else**
11: $(valid, \mathcal{M}) \leftarrow MAX_{valid}(\{(valid, \mathcal{M}) | valid \in [\frac{s}{2}, s] \wedge F_{est} \sim_{\mathcal{M}} \theta_{valid}\})$
12: **if** $valid \leq 1$ **then** $state \leftarrow F_{mkv}(state)$; $valid \leftarrow 1$
13: **else** $state \leftarrow \mathcal{M}(F_{est}(state))$
14: $target \leftarrow target - valid$; $s \leftarrow 1$;
15: $\theta_1 \leftarrow state[0 \ldots m]$; $F_{est} \leftarrow F_{mkv}(\theta_1)$

the full set of instances being generated. At each step, the fingerprint of the Markov function is compared to that of the estimator. The number of steps between comparisons grows exponentially until (4.b) the algorithm finds a mismatch. (4.c) At this point, the algorithm backtracks to the last matching value with a binary search and uses the estimator to regenerate the full state of the Markov process. (4.d) The Markov process is used to step the full set of instances until the estimator function once again begins to produce matching fingerprints.

4.2 Generating Estimator Functions

The user does not need to explicitly provide an estimator function. Simple cyclical dependencies between models allows us to extract an estimator function by fixing one model's output to its value at a given step. Indeed, any Markov function that models an infrequently discontinuous process can be made into a viable estimator by reusing state in a similar way.

A function F_{mkv} defining a Markov process with per-step state P_i generates the next step's state: $F_{mkv}(P_i, Q) = P_{i+1}$. We can define a rudimentary estimator function $F_{est,i}$ by fixing F_{mkv}'s input state at one point in time.

$$F_{est,i}(Q) = F_{mkv}(P_i, Q)$$

Even this rudimentary estimator function can be quite powerful when combined with fingerprints; any uniform changes in state are absorbed by the mapping function.

For example, consider the Markov jump query illustrated in Figure 5. The special `CHAIN` parameter type is used to chain the output of one stage of the Markov computation to the following one – in this case chaining the output of `ReleaseWeekModel` to the subsequent `DemandModel` invocation.

As before, `ReleaseWeekModel` has a single discontinuity at the point where `DemandModel`'s output exceeds a certain threshold. Each step in the Markov chain corresponds to

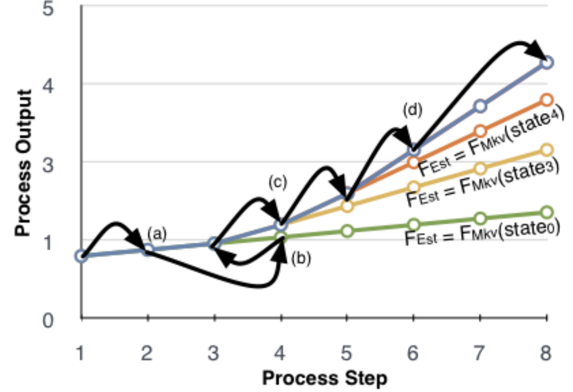

Figure 4: An example execution of the Markov Jump algorithm. The algorithm starts by performing a fingerprint-sized sampling (a) from the markov chain, until the fingerprint differs from a synthesized estimator (b), then backtracks and synthesizes a new estimator (c). This process repeats until it finds an estimator valid for the remainder of the chain (d).

```
-- DEFINITION --
DECLARE PARAMETER @current_week
        AS RANGE 0 TO 52 STEP BY 1;
DECLARE PARAMETER @release_week
        AS CHAIN release_week
        FROM @current_week : @current_week - 1
        INITIAL VALUE 52;
SELECT ReleaseWeekModel(demand) AS release_week, demand
  FROM (SELECT DemandModel(@current_week, @release_week)
               AS demand)
  INTO results
-- BATCH MODE --
...
```

Figure 5: A Jigsaw query with a Markovian dependency

predictions for one specific week. The interesting output of this model is `demand`. An estimator from this value will be constructed by fixing `release_week` (the chain parameter) at its initial value. Until the markov process enters the region of the chain (and after it exits) where the discontinuity is likely to occur, the demand model can be effectively approximated by this non-Markovian estimator.

5. INTERACTIVE WHAT-IFS

Jigsaw's heuristic approach to sampling is ideally suited to the task of online what-if exploration. Moreover, the sort of parameter exploration problems that Jigsaw addresses also benefit from having a human in the loop—imprecise goal conditions that are difficult to specify programmatically can often be reached easily by an expert human operator.

A human operator indicates which regions of the parameter space are interesting, and Jigsaw provides progressively more accurate results for that region. Metadata supplementing the simulation query allows Jigsaw to interpret the query results and to produce and progressively refine a graphical representation of the query output for a given set of parameter values.

Unlike its offline counterpart, the goal of online Jigsaw is to rapidly produce accurate metrics for a small set of points in the parameter space. Fingerprinting is used primarily to

Algorithm 5 *SimplifiedEventLoop*$(p, State)$

Require: One point of interest p. A lookup table $State[]$ containing, for all points: a mapping function $\mathcal{M}$, the point's fingerprint θ, and the point's basis distribution.

1: **loop**
2: $(\theta, basis, \mathcal{M}) \leftarrow State[p]$
3: $next \leftarrow p$
4: $task \leftarrow TaskHeuristic(p)$
5: **if** $task =$ refinement **then**
6: $candidate_ids \leftarrow \{id | id \notin basis\}$
7: **else if** $task =$ validation **then**
8: $candidate_ids \leftarrow \{id | id \in basis \wedge id \notin \theta\}$
9: **else if** $task =$ exploration **then**
10: $next \leftarrow ExploreHeuristic(p)$ {Find a nearby point}
11: **if** $State[next].\theta \neq \emptyset$ **then**
12: $candidate_ids \leftarrow \{id | id \notin State[next].basis\}$
13: **else**
14: $candidate_ids \leftarrow [0, 10)$
15: $sample_ids \leftarrow PickAtRandom(10, candidate_ids)$
16: $values \leftarrow EvaluateBlackBox(next, sample_ids)$
17: $State[next].\theta \leftarrow State[next].\theta \cup values$
18: **if** $State[next].basis \sim_{\mathcal{M}} State[next].\theta$ **then**
19: $(State[next].basis, State[next].\mathcal{M}) \leftarrow FindMatch(State[next].\theta)$
20: **else**
21: $State[next].basis \leftarrow State[next].basis \cup \mathcal{M}^{-1}(values)$

improve the accuracy of Jigsaw's initial guesses; a very small and quickly generated (e.g., of size 10) fingerprint allows Jigsaw to identify a matching basis distribution and reuse metrics precomputed for it.

Jigsaw provides the following three categories of processing tasks:

Refinement. Once the initial guess is generated, Jigsaw begins generating further samples for points (i.e., parameter values) of interest. In addition to improving the accuracy of the displayed results, the new samples are used to improve the accuracy of the basis distribution's precomputed metrics.

Validation. Latency also places stringent requirements on the size of fingerprints. Larger fingerprints produce more accurate estimates, but take longer to produce. However, in an online setting, Jigsaw constructs the fingerprint progressively. In addition to generating additional samples for the basis distribution, Jigsaw also reproduces samples for the points of interest that are already present in the basis distribution. The duplicate samples effectively extend the point's fingerprint by validating the existing mapping; if the new points do not match the values mapped from the basis distribution, Jigsaw finds or creates a new basis distribution.

Exploration. In addition to the above two processing tasks, Jigsaw heuristically selects points in the parameter space that are likely to be of interest to the user in the near future (e.g., adjacent points in a discrete parameter space). For each point explored, Jigsaw either generates a fingerprint (if none exists), or extends the point's basis distribution with a small number of additional samples.

For clarity, we have drawn a distinction between samples produced for fingerprints and those produced for basis distributions. However, in most cases there is no difference between either process. For any invertible mapping function, samples are generated directly for the point of interest, and mapped back to the basis distribution by the inverse of the mapping function $\mathcal{M}^{-1}$. For example, for linear mappings $\mathcal{M}(x) = \alpha x + \beta$, the inverse $\mathcal{M}^{-1}(x) = \frac{x-\beta}{\alpha}$.

The core of online Jigsaw is a relatively simple pick-evaluate-update process: (1) Pick the next set of (point, sampleID) pairs to generate samples for, (2) Evaluate the query, and (3) Update the fingerprint, basis, and mapping. This process is presented in Algorithm 5.

6. EXPERIMENTS

In this section we experimentally evaluate effectiveness of various optimizations used in Jigsaw.

Implementations. The original prototype of Jigsaw is implemented as a C# PDB layer built on top of Microsoft SQL Server. The black box functions are implemented as stored procedure written in C#. The C# layer interacts with the SQL Server query execution engine by simply invoking it on subqueries and post-processing the results outside DBMS. However, this implementation is not well-suited for performance evaluation of Jigsaw, as timing results are polluted by noise from interprocess communication and SQL interpretation and evaluation overheads. In order to achieve a more representative comparison, and to streamline the testing process, we have constructed a second prototype of Jigsaw query evaluation engine entirely in Ruby (without any commercial DBMS). Queries are implemented as black box functions in Ruby, and invoked by a driver process. This simple implementation is representative of how Jigsaw's functionality can be implemented within a probabilistic database's query evaluation engine. We compare these two prototypes in Section 6.1.

Black Box Functions. Experiments use a variety of black boxes described in Figure 6. Though several synthetic black-boxes are used to identify specific performance characteristics, the Capacity, Demand, Overload, User Selection and Markov Step black boxes are permutations of actual Jigsaw use cases in real cloud infrastructure management scenarios. Specific numbers (i.e., the mean and standard deviation of a normal distribution) have been replaced by ad-hoc values, but the structure of these models remains intact. In all experiments, we explore the entire parameter space for particular black boxes.

Experimental Setup. Experiments are performed on a 2.4 GHz Core2 Duo with 4 GB of RAM. Except where stated, experiments assume a need for exactly 1000 sample instances per point in the parameter space, and use a fingerprint size of 10. All results shown are the average of 30 trials.

6.1 Comparison of Two Prototypes

Figure 7 shows a brief comparison of the relative performance of Jigsaw's C# + MS SQL implementation and the lightweight Ruby engine. As shown, for simple data-independent queries, the Ruby implementation is able to achieve much better performance, as the dominant cost is that of invoking the black box, rather than the overheads of repeatedly invoking the query processor. The one case where the ruby implementation is not representative of the offline engine is black boxes that are heavily data dependent, as in the UserSelection simulation. As might be expected, a

Capacity(current_date, purchase_date_1, purchase_date_2). The Capacity black box simulates a series of purchases. Each purchase increases the capacity of the server cluster after an exponentially distributed delay.

Demand(current_date, feature_release). The Demand black box simulates a simple linearly growing gaussian demand model. As of the feature_release week, the growth rate is changed.

Overload(current_date, purchase_date_1, purchase_date_2). A black box synthesized from Capacity and Demand. Demand's feature_release is ignored, and this black box returns 1 if Demand is greater than Capacity, and 0 otherwise.

UserSelection(current_date). The UserSim black box simulates the per-user requirements of each of a set of users.

SynthBasis(parameter_point). A synthetic black box based on Demand, but with a deterministic number of basis distributions.

MarkovStep(current_date, before_or_after). A simple Markovian process simulating the behavior of Demand with a Markovian dependency introduced between feature_release and the prior date's demand.

MarkovBranch(prior_state). A synthetic black box where at each step, a state counter is incremented by one with a predefined probability. The states diverge at some specified rate.

Figure 6: Black boxes used to evaluate Jigsaw

Model	Online Speed	Offline Speed
Demand	0.1964 s/pc	0.00096 s/pc
Capacity	0.84525 s/pc	0.0028 s/pc
Overload	5.4625 s/pc	0.092825 s/pc
UserSelect	34.4 s/pc	252.454 s/pc

Figure 7: User Interface Wrapper vs Core Engine Simulator Timing comparison. Values are in time per parameter combination.

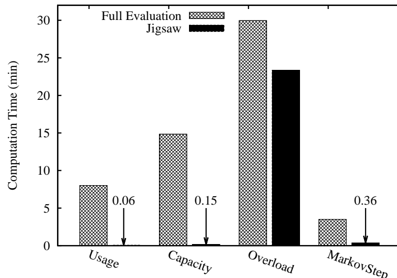

Figure 8: Jigsaw vs fully exploring the parameter space.

database engine's ability to manage large datasets is superior to that of Ruby.

The rest of the experiments in this section use less-data dependent black boxes, and hence we use the Ruby implementation. However, relative performance gains demonstrated by the Ruby prototype are roughly similar to that in the C# + MS SQL implementation.

6.2 Baseline Performance

We now analyze the standalone performance gain of fingerprinting by comparing against a naive-generate everything approach. Figure 8 shows timing results for several queries, each evaluated both with and without fingerprinting. The extremely simplistic *Demand* model requires only one basis distribution for its entire ∼5000 point parameter space, and can be evaluated almost instantantaneously. Even relatively complex event-based models like *Capacity* (which has a parameter space of ∼8000 points) and *MarkovStep* (evaluated over ∼2500 steps) require only a few basis distributions.

Overload is an interesting case: despite being defined as a query over two two black boxes for which Jigsaw can provide a substantial performance boost (compare Overload

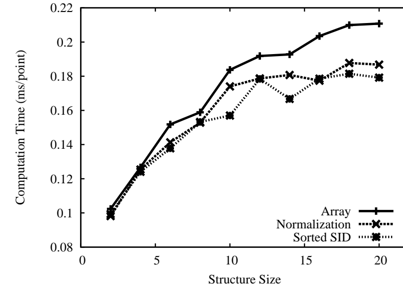

Figure 9: Computation time versus the size of structures in the Capacity model

with ∼8000 points and Demand and Capacity's timing), the joint query is only computed in half the time. The reason for this is that the query produces a boolean result: the output of a comparison between two values, and information about the two original values is lost. Effectively, Jigsaw is unable to reuse basis values by re-mapping them. This strongly suggests that Jigsaw's techniques can be further improved by incorporating them into a database engine with a symbolic execution strategy (e.g. [13]). In such a system, database operations between random variables (i.e., VG-Function-generated values) mapped from the same basis distribution are resolved symbolically. For example, consider two random variables X, Y such that $X = \mathcal{M}_X(f(x)) = 2 \cdot f(x) + 2$ and $\mathcal{M}_Y(f(x)) = 3 \cdot f(x) + 3$. We can symbolically produce $X + Y = (\mathcal{M}_X + \mathcal{M}_Y)(f(x)) = 5 \cdot f(x) + 5$. Similarly, given a histogram of $f(x)$ we can efficiently compute the probability that $M_X(f(x)) > M_Y(f(x))$.

The *Capacity* model also deserves more discussion. Recall the overview provided in Section 2. As discussed in Section 2.2, the model produces a line with several non-localized discontinuities or *structures*, one for each purchase. However, each of the discontinuities is surrounded by a structure spanning a range of dates: each purchase is followed by a short period during which the simulated hardware has not come online in a(n exponentially shrinking) fraction of the sample instances. Figure 9 relates the number of basis distributions to the size of each structure (the number of "weeks" that it spans). Note that the relationship between structure size grows and the number of basis distributions is sub-linear. Jigsaw is able to recognize individual points in each structure (i.e., four weeks after one purchase, and the week of the second purchase), and reuse the corresponding basis distribution.

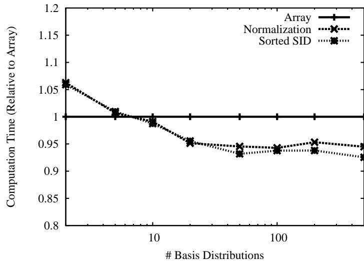

Figure 10: Indexing in a static parameter space.

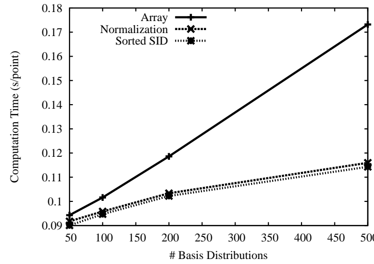

Figure 11: Indexing, growing the parameter space with basis size.

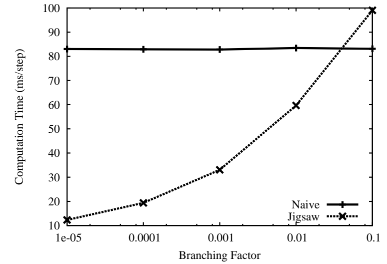

Figure 12: Performance for a Markov process.

Accuracy. In theory, our principle of using fingerprints can introduce two sources of errors in a general simulation framework. The first source is the possibility of selecting an incorrect fingerprint due to insufficient fingerprint length. We have not observed any significant error of this sort in any of our our experiments, suggesting that a fingerprint length of 10 is sufficient for the models we consider. The second source of possible error is due to correlation of results under different parameter values (since we use the same random seed across parameter values). However, Jigsaw never combines such correlated results of different parameters—it only compares them. Hence we do not see such errors in our results. In other words, outputs of Jigsaw are equivalent to full simulation for each possible parameter value.

6.3 Indexing

We now explore the behavior of the two indexing strategies described in Section 3:*Normalization*, and *Sorted SID* indexing. In this test, we synthesized black boxes, each designed to generate a specific number of basis distributions. We computed the expectation of each black-box for 1000 different parameter combinations. Figure 10 shows the performance of each indexing strategy relative to performing a naive *Array* scan for each lookup. The overall results are not surprising: The costs of an array scan begin to dominate the static costs of indexing as the basis grows past 50 elements.

Both indexing schemes perform substantially better than naive array scan, with Normalization trailing behind Sorted SID slightly. After we reach a basis size of about 200 (full sample generation is required for 20% of the parameter space), the cost of sample generation begins to dominate the cost of basis matching. Indexing continues to asymptotically approach a 10% reduction in computation time. This is best illustrated in Figure 11. Here, we scale the size of the parameter space (and consequently the total amount of computation) relative to the basis size. We fix the basis size at 10% of the parameter space. As expected, naive Array scan scales linearly with basis size, while the indexing strategies scale sub-linearly.

6.4 Markovian Jumps

Next, we analyze Jigsaw's performance on Markov processes. Markov processes consisting of periodic, but infrequent discontinuities are ideal for Jigsaw; this sort of black box behavior generates frequently overlapping states and admits an easy estimator, which simply assumes that the state stays the same. Such a process has already been illustrated in Figure 8.

We now consider the benefits and limitations of Jigsaw on more complex, diverging models. Figure 12 shows Jigsaw's performance on a black-box Markovian process synthetically generated to diverge at a predefined rate. We use the term branching to refer to the probability of divergence at each timestep. The black box was invoked for 128 steps, and Jigsaw attempted to accelerate evaluation by skipping ahead in the process.

This shows Jigsaw's applicability to Markovian processes characterized by periodic discontinuities. Even in its default configuration, Jigsaw is able to improve the efficiency of Markovian processes where as many as one in twenty steps involves a discontinuity. Indexing strategies designed specifically for Markovian processes (e.g., discard all basis values except the last), can improve this even further.

7. RELATED WORK

Jigsaw is designed to act as a component of a probabilistic database system [3, 4, 12, 13, 16, 18, 5]. These systems make probability distributions into a directly representable and queryable datatype. Most probabilistic database systems restrict their users to either discrete probability distributions, and/or a carefully selected set of well known continuous probability distributions. Orion 2[18] takes this one step further and allows the use of arbitrary distributions, if it can evaluate its PDF. However, two systems: MCDB [12] and PIP [13] aim for a much general distribution support. In these systems, users specify distributions through stochastic black-box functions (commonly referred to as variable generating, or VG-Functions). This extremely general approach is necessary for Jigsaw's application domain. Jigsaw builds on their functionality, adding support for efficient parameter exploration and optimization tasks.

MCDB focuses on constructing an efficient infrastructure for black-box invocation and query evaluation. Recent work in this space explores the use of parallel processing [20] and different sampling techniques for highly selective queries [10], but does not exploit any analytically useful properties of individual black-box functions. Conversely, Jigsaw's ability to identify correlations between different parameterizations of a black-box function makes it more efficient at queries that enumerate all possible parameter combinations.

PIP employs a grey-box technique, where developers have the option of specifying characteristics of their functions with supplemental metadata. However, this additional metadata must be provided in order to obtain any performance benefits. Jigsaw is able to infer function characteristics without programmer support, allowing it to present a much simpler programming interface.

A related area of database research involves representing continuous functions as tables. Pulse [1], MauveDB [6], and

FunctionDB [19] allow users to construct functional models within the database. Queries posed over these models are evaluated symbolically to the extent possible, substantially improving performance. However, these systems necessitate a functional representation of the data being modeled, and thus lack the generality of VG-Functions. However, once Jigsaw has extracted a set of mapping functions $\mathcal{M}$ for an entire parameter space, symbolic querying techniques such as these could be applied to improve performance further.

Functional representation of stochastic black-boxes is a field that has been explored extensively. Techniques ranging from simple curve-fitting, wavelets [9], various space transforms [2], and even simple hat functions [7] have been proposed as mechanisms for producing functional representations of stochastic black-boxes.

A number of techniques [17, 14] have also been developed for performing optimization over black-box functions. However, all of these techniques are developed for uncontrollable black-boxes, typically real-world processes. Consequently, they are limited to a regression-style approach. Conversely, Jigsaw controls the source of randomness within the function, which allows it to deterministically generate fingerprints.

Constrained optimization has also been considered in the context of databases [8], but with a view towards minimizing IO requirements. Indexes containing data bounds are used to prune the search space. However, this approach relies on the continuity of the function being optimized; VG-Functions negate this assumption.

The integration of specialized modeling tools with database systems [21] has also been explored. However, while such tools streamline the process of fitting a model, they do not focus on model evaluation. One could imagine such a tool being integrated into the Jigsaw workflow for the construction of VG-Functions.

8. CONCLUSIONS

In this paper, we have demonstrated Jigsaw, a powerful tool for evaluating and optimizing parameterized what-if scenarios. Jigsaw efficiently performs parameter optimization, allows online exploration of a scenario's parameter space at interactive speeds, and rapid evaluation of a common class of Markovian processes. The key to these three processes is a novel "fingerprinting" mechanism which identifies correlations between similar, yet distinct probability distributions. We have shown that fingerprints can be applied to several common tasks that arise in the domain of cloud service management, and demonstrated that Jigsaw can achieve speedups of as much as 2 orders of magnitude.

Acknowledgements. The authors would like to acknowledge Steve Lee, Charles Loboz, and Slawek Smyl from Microsoft Windows Azure for providing motivating example scenarios and technical feedback, as well as Christoph Koch from EPFL for providing support and technical feedback.

9. REFERENCES

[1] Y. Ahmad, O. Papaemmanouil, U. Çetintemel, and J. Rogers. Simultaneous equation systems for query processing on continuous-time data streams. In *ICDE*, pages 666–675. IEEE, 2008.

[2] N. Ahmed, T. Natarajan, and K. Rao. Discrete cosine transfom. *Computers, IEEE Transactions on*, C-23(1):90 – 93, jan. 1974.

[3] L. Antova, C. Koch, and D. Olteanu. MayBMS: Managing incomplete information with probabilistic world-set decompositions. In *ICDE*, 2007.

[4] J. Boulos, N. N. Dalvi, B. Mandhani, S. Mathur, C. Ré, and D. Suciu. MYSTIQ: a system for finding more answers by using probabilities. In *ACM SIGMOD*, 2005.

[5] P. Cudré-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. Wang, M. Balazinska, J. Becla, D. J. DeWitt, B. Heath, D. Maier, S. Madden, J. M. Patel, M. Stonebraker, and S. B. Zdonik. A demonstration of SciDB: A science-oriented DBMS. *PVLDB*, 2(2):1534–1537, 2009.

[6] A. Deshpande and S. Madden. MauveDB: supporting model-based user views in database systems. In *ACM SIGMOD*, 2006.

[7] L. Devroye. *Non-Uniform Random Variate Generation*. Springer-Verlag, New York, 1986.

[8] M. Gibas, N. Zheng, and H. Ferhatosmanoglu. A general framework for modeling and processing optimization queries. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 1069–1080. VLDB Endowment, 2007.

[9] A. Haar. Zur theorie der orthogonalen funktionensysteme. *Mathematische Annalen*, 69:331–371, 1910. 10.1007/BF01456326.

[10] P. J. Haas, C. M. Jermaine, S. Arumugam, F. Xu, L. L. Perez, and R. Jampani. MCDB-R: Risk analysis in the database. *PVLDB*, 3(1):782–793, 2010.

[11] D. Hamlet. *Encyclopedia of Software Engineering*, chapter Random testing, pages 970–978. Wiley, New York, 1994.

[12] R. Jampani, F. Xu, M. Wu, L. L. Perez, C. M. Jermaine, and P. J. Haas. MCDB: a monte carlo approach to managing uncertain data. In *ACM SIGMOD*, 2008.

[13] O. Kennedy and C. Koch. PIP: A database system for great and small expectations. In *ICDE*, 2010.

[14] H. J. Kushner and D. S. Clark. *Stochastic approximation methods for constrained and unconstrained systems*. Springer-Verlag, 1978.

[15] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

[16] M. Mutsuzaki, M. Theobald, A. de Keijzer, J. Widom, P. Agrawal, O. Benjelloun, A. D. Sarma, R. Murthy, and T. Sugihara. Trio-One: Layering uncertainty and lineage on a conventional dbms (demo). In *CIDR*, pages 269–274, 2007.

[17] M. H. Safizadeh and B. M. Thornton. Optimization in simulation experiments using response surface methodology. *COMP. INDUST. ENG.*, 8(1):11–28, 1984.

[18] S. Singh, C. Mayfield, S. Mittal, S. Prabhakar, S. E. Hambrusch, and R. Shah. Orion 2.0: native support for uncertain data. In *ACM SIGMOD*, 2008.

[19] A. Thiagarajan and S. Madden. Querying continuous functions in a database system. In *ACM SIGMOD*, 2008.

[20] F. Xu, K. S. Beyer, V. Ercegovac, P. J. Haas, and E. J. Shekita. $E = mc^3$: managing uncertain enterprise data in a cluster-computing environment. In *ACM SIGMOD*, 2009.

[21] Y. Zhang, W. Zhang, and J. Yang. I/O-efficient statistical computing with RIOT. In F. Li, M. M. Moro, S. Ghandeharizadeh, J. R. Haritsa, G. Weikum, M. J. Carey, F. Casati, E. Y. Chang, I. Manolescu, S. Mehrotra, U. Dayal, and V. J. Tsotras, editors, *ICDE*, pages 1157–1160. IEEE, 2010.