The Monte Carlo Database System: Stochastic Analysis Close to the Data

RAVI JAMPANI, University of Florida
FEI XU, Microsoft Corporation
MINGXI WU, Oracle Corporation
LUIS PEREZ and CHRIS JERMAINE, Rice University
PETER J. HAAS, IBM Almaden Research Center

The application of stochastic models and analysis techniques to large datasets is now commonplace. Unfortunately, in practice this usually means extracting data from a database system into an external tool (such as SAS, R, Arena, or Matlab), and then running the analysis there. This extract-and-model paradigm is typically error-prone, slow, does not support fine-grained modeling, and discourages what-if and sensitivity analyses.

In this article we describe MCDB, a database system that permits a wide spectrum of stochastic models to be used in conjunction with the data stored in a large database, without ever extracting the data. MCDB facilitates in-database execution of tasks such as risk assessment, prediction, and imputation of missing data, as well as management of errors due to data integration, information extraction, and privacy-preserving data anonymization. MCDB allows a user to define "random" relations whose contents are determined by stochastic models. The models can then be queried using standard SQL. Monte Carlo techniques are used to analyze the probability distribution of the result of an SQL query over random relations. Novel "tuple-bundle" processing techniques can effectively control the Monte Carlo overhead, as shown in our experiments.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Relational databases*; I.6.7 [**Simulation and Modeling**]: Simulation Support Systems

General Terms: Algorithms, Performance

Additional Key Words and Phrases: MCDB, relational database systems, uncertainty

ACM Reference Format:
Jampani, R., Xu, F., Wu, M., Perez, L., Jermaine, C., and Haas, P. J. 2011. The Monte Carlo database system: Stochastic analysis close to the data. ACM Trans. Datab. Syst. 36, 3, Article 18 (August 2011), 41 pages.
DOI = 10.1145/2000824.2000828 http://doi.acm.org/10.1145/2000824.2000828

1. INTRODUCTION

Analysts often need to work with large datasets in which some of the data is uncertain, often because the data is connected to hypothetical or future events. For example, the data of interest might be the customer order sizes for some product under a hypothetical

This work was supported by the National Science Foundation under grant 0915315, by the Department of Energy under grant DE-SC0001779, and by a Sloan Foundation Fellowship.

Authors' addresses: R. Jampani, University of Florida, Gainesville, FL 32611; F. Xu, Microsoft Corporation, One Microsoft Way, Redmond, WA 98052-6399; M. Wu, Oracle Corporation, 500 Oracle Parkway, Redwood Shores, CA 94065; L. Perez and C. Jermaine (corresponding author), Computer Science Department, Rice University, 6100 Main, Houston, TX 77005-1827; email: cmj4@cs.rice.edu; P. J. Haas, IBM Almaden Research Center, 1 New Orchard Road, Armonk, NY 10504-1722.

DOI 10.1145/2000824.2000828 http://doi.acm.org/10.1145/2000824.2000828

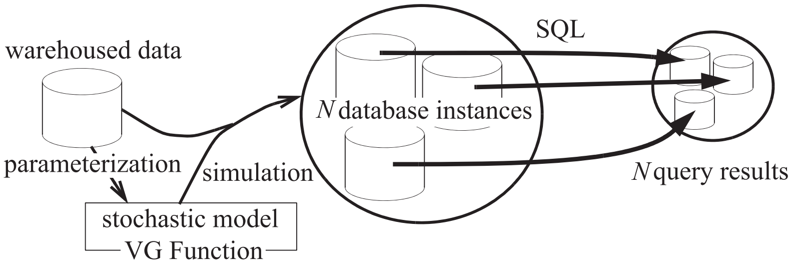

Fig. 1. Basic query processing framework supported by MCDB.

price increase of 5%. Data uncertainty is frequently modeled as a probability distribution over possible data values. Such probabilities are usually specified by means of a complex stochastic model, for example, a stochastic demand model that predicts the probability distribution of a customer's order size at a given price level. Various system characteristics of interest to the analyst can be viewed as answers to queries over the uncertain data values. In our example, the total product revenue under the stochastic demand model is computed by multiplying each customer's (uncertain) order quantity by the price and summing over all customers. Because the data is uncertain, there is a probability distribution over the possible results of running a given query, and the analysis of the underlying stochastic model amounts to studying the features (mean, variance, and so forth) of the query-result distribution. The query-result distribution is often very complex, and must be analyzed using Monte Carlo methods. Such analysis is key to assessing enterprise risk, as well as making decisions under uncertainty.

This article introduces a prototype relational DBMS called the *Monte Carlo Database System*, or *MCDB*. MCDB allows an analyst to attach arbitrary stochastic models to a database, thereby specifying, in addition to the ordinary relations in the database, "random" relations that contain uncertain data. These stochastic models are implemented as user- and system-defined libraries of external C++ programs called *Variable Generation functions*, or VG functions for short. A call to a VG function generates a pseudorandom sample from a data-value probability distribution; VG functions are usually parameterized on the current state of the nonrandom relations (tables of historical sales data in our example). Generating a sample of each uncertain data value in the database creates a *database instance*, that is, a realization of an ordinary database. Running a SQL query over the database instance generates a sample from the query-result distribution. Iterating this process N times generates N samples from this latter distribution, which can then be used to estimate distribution features of interest. The process is illustrated in Figure 1. In our "price increase" example, we might estimate the expected value of the total revenue by the average of the total revenue values over the N query results.

Motivation: Why Does Anyone Need MCDB? Data models and databases for uncertain and/or probabilistic data are a hot topic in data management research; see Section 2. MCDB is related to the extent that the goal is to incorporate management of uncertainty into the database system. Thus, a reasonable question is: why do we need another system designed for the management of uncertain data?

In fact, MCDB is quite different from these systems in terms of motivation, design, and application space; we would not call MCDB a "probabilistic database system." We elaborate on this point in Section 2. It is more useful to conceive of MCDB as an alternative for the following workflow often found in large data warehouse installations.

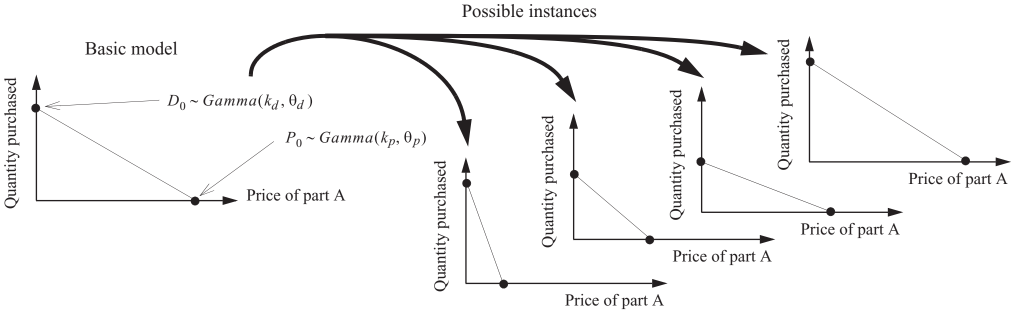
Possible instances
Basic model
$D_0 \sim Gamma(k_d, \theta_d)$
$P_0 \sim Gamma(k_p, \theta_p)$
Quantity purchased
Price of part A

Fig. 2. Stochastic generation of customer demand functions.

(1) First, an analyst (perhaps a PhD-level statistician or econometrician) decides to use the data stored in the warehouse to make some sort of statistical prediction based on both the warehoused data and additional data values that are unknown or unavailable, hence uncertain. In our example, the goal would be to use the warehoused data to predict what a part supplier's revenue would have been last year, had the supplier's prices for a given part been increased by 5%.

(2) A stochastic model for the uncertain data is then conceived. In our example, one might define a model where a simple linear demand curve is generated by sampling the demand at a price of zero from a Gamma distribution; that is, $D_0 \sim \text{Gamma}(k_d, \theta_d)$, where k_d and θ_d are the "shape" and "scale" parameters. Similarly, the price at which zero units are purchased is $P_0 \sim \text{Gamma}(k_p, \theta_p)$. This is exemplified in Figure 2. Under this model of random demand curves, each fixed price yields a probability distribution over the order amount.

(3) Data (or appropriate summaries of the data) are then extracted from the database, and used to train or tailor the model to the data in the database. In our example, this might take the form of "learning" the four parameters k_d, θ_d, k_p, and θ_p—perhaps using maximum likelihood estimation [Lehmann and Casella 1998] based on the historical data presently in the database. In the parlance of Bayesian statistics [O'Hagan and Forster 2004], the result of the learning step is a "prior" distribution on the customer's demand curve for a specific item. This learning typically takes place in an external tool, such as R or Matlab.

(4) In combination with the database data, the trained model is used to answer the original question. In our example, this might require going through each of the sales transactions for 2010 that are stored in the database. For an observed sale of d^* units of a given part at price p^*, we want to guess the number of units D_p that *would* have been sold at the new price p. To do this, we sample a (D_0, P_0) pair from the "posterior" distribution.

$$(D_0, P_0) \sim ((\text{Gamma}(k_d, \theta_d), \text{Gamma}(k_p, \theta_p)) \mid (D_0/P_0)(P_0 - p^*) = d^*)$$

In Bayesian parlance, the prior distribution on the demand curve has been updated to take into account the known fact that the demand curve passes through the point (p^*, d^*). Once the pair (D_0, P_0) has been sampled, then D_p can be computed as $D_p = (D_0/P_0)(P_0 - p)$. After generating the new sales quantity for every 2010 sale, a new total revenue value is computed. Iterating this process N times, we obtain N samples from the total revenue distribution.

Using current technology for such a workflow is exceedingly problematic for several reasons.

(1) The process of writing code to extract data from the database in order to train the model is buggy, error-prone, and slow. This difficulty in moving data between the database and the analysis environment discourages sensitivity and what-if analysis for the model.
(2) Because the model is external to the database, the model's level of granularity is limited. In fact, realizing the example workflow described before is likely impossible for a large organization, because one cannot apply the learned models 100 million times to 100 million different prior-year sales that are stored in the database. Extracting hundreds of millions of records from a database system to feed models that live in Matlab is quite unrealistic. Instead, the model must be applied at a coarser level of granularity, for example, by lumping customers together, and thereby losing predictive power.
(3) There is no encapsulation of the model. In the revenue prediction scenario, for example, it is not possible to attach the model to the database and allow subsequent queries from a nonexpert analyst who does not understand stochastic models but does understand demand functions. The PhD-level statistician or econometrician who defined the model must either leave detailed instructions on how to repeat the training/usage workflow, or (more likely) be involved in every subsequent application of the model.
(4) One cannot easily "relink" the results of the modeling process back to the existing data. In our example, it would be difficult to use the new, simulated sales to compute new profits, because this computation requires joining the modeling results with cost information that is only available back in the database.
(5) Because it is external to the database, the model must be reparameterized every time the data are updated.

MCDB addresses the preceding shortcomings by pushing the entire stochastic modeling process (both training, and application) down into the database system. This simplifies model development, testing, training, and deployment, and facilitates sensitivity analysis for assessing the robustness of model results to model assumptions.

Our Contributions. The article's contributions are as follows.

—We propose the first "pure" Monte Carlo approach toward managing complex models of uncertain data. Although others have suggested the possibility of Monte Carlo techniques in probabilistic databases [Ré et al. 2007], ours is the first system for which the Monte Carlo approach is fundamental to the entire system design.
—We propose a powerful and flexible representation of data uncertainty via schemas, VG functions, and parameter tables (Section 4).
—We provide a syntax for specifying the use of random relations that requires only a slight modification of SQL, and hence is easily understood by database programmers (Section 4). The specification of VG functions (Section 5) is very similar to specification of User-Defined Functions (UDFs) in current database systems.
—To ensure acceptable practical performance, we provide new query processing algorithms (Sections 6, 8, 9, and 10) that execute a query plan only once, processing "tuple bundles" rather than ordinary tuples. A tuple bundle encapsulates the instantiations of a tuple over a set of Monte Carlo iterations (Section 7). We exploit properties of pseudorandom number generators to maintain the tuple bundles in highly compressed form whenever possible.
—We show, by running a collection of interesting benchmark queries on our prototype system, that MCDB can provide novel functionality with acceptable performance overheads (Section 13).

This article, an extended version of Jampani et al. [2008], focuses on the overall design of MCDB, potential applications, and the implementation of the query-processing engine. Issues of query plan generation and optimization—which are being addressed in ongoing work—will be discussed in a subsequent paper. (Section 11.4 touches briefly on this topic.)

The new material in this article includes additional examples of MCDB usage (Section 4), discussion of additional relational operations (Section 9), a detailed example of query evaluation in MCDB (Section 10), a discussion of implementation details such as random variate generation and query plan manipulation (Section 11), as well as how the accuracy of inferences made using MCDB might be controlled by the user (Section 12). In addition, seven of the eleven benchmarking/demonstration queries are new (Section 13). An online appendix contains material from Jampani et al. [2008] that was excluded here due to space constraints, as well as a formal grammar for MCDB SQL and additional examples of the use of MCDB for statistical estimation. The online appendix can be found in the ACM Digital Library.

2. RELATED WORK

The MCDB system was inspired by several efforts to push analytic capabilities into the database (or, more generally, "close to the data"). Some very simple statistical regression technology has been natively supported by relational database systems for about a decade, and has been incorporated into the SQL standard [Alur et al. 2002]. A more recent effort, and a major inspiration for our work, is MauveDB [Deshpande and Madden 2006], which supports "model-based views" over streaming sensor data. These ideas have been further developed in subsequent systems such as FunctionDB [Thiagarajan and Madden 2008], which allows native querying of statistical regression functions, and the recent SciDB project [Stonebraker et al. 2009]. SciDB overhauls the relational data model—using multidimensional arrays for storage and making vectors and arrays first-class objects—and executes functions and procedures in parallel and as close to the data as possible. A recent discussion of analytics in the database can also be found in Cohen et al. [2009], where matrix operations in a relational database are implemented using UDFs. There have been a number of efforts to push various kinds of statistical analysis into Map-Reduce data processing environments; see, for example, ApacheMahout [2010], Chu et al. [2006a], and Guha [2010]. All of this work has primarily focused on interpolation and smoothing, statistical model fitting, and data mining. In contrast, MCDB is focused on Monte Carlo analysis of stochastic models; indeed, MCDB is the first DBMS for which the Monte Carlo approach is fundamental to the entire system design.[1]

As mentioned previously, MCDB is also related to work on probabilistic database systems (ProbDBs), although MCDB is not primarily designed to be used as such a system. Work in ProbDBs dates back at least to the 1990's [Barbará et al. 1992; Fuhr and Rölleke 1997], and the past few years have seen a great deal of research on this topic. In particular, a number of ProbDB prototypes have been developed, including Trio [Agrawal et al. 2006; Das Sarma et al. 2008], MayBMS [Antova et al. 2007, 2008; Koch and Olteanu 2008], MystiQ [Ré and Suciu 2008; Wang et al. 2008b], ORION [Cheng et al. 2005; Singh et al. 2008], PrDb [Sen et al. 2009], and BayesStore [Wang et al. 2008a]. Some recent overviews of the area are given in Benjelloun et al. [2008], Dalvi and Suciu [2007c], Dalvi et al. [2009], and Das Sarma et al. [2009], and extensions to uncertain XML data can be found in Kimelfeld et al. [2009].

[1] The recent PIP system of Kennedy and Koch [2010] combines PrDB and Monte Carlo techniques, and can yield superior performance for certain MCDB-style queries focusing on expected values and having simple VG-function parameterizations.

Like MCDB, ProbDBs assume a probability distribution over uncertain data values, and hence over possible DBs ("possible-world semantics"). Unlike MCDB's emphasis on complex (often predictive) stochastic models, however, ProbDBs focus on handling uncertainty that is "lurking in the data warehouse," typically arising from sources such as integration of inconsistent or dirty data [Andritsos et al. 2006; Dong et al. 2009], information extraction from text [Gupta and Sarawagi 2006; Michelakis et al. 2009], and noisy sensor readings [Chu et al. 2006b]. Consequently, the most common "stochastic model" simply comprises, for each tuple, a discrete probability distribution over possible attribute values for the tuple (including the possibility that the tuple is not present in the relation). These probabilities are stored with the tuple, and are treated as first-class objects; a typical example of such an extended tuple is an "x-tuple" in the Trio system. (In MCDB, such probabilities would be viewed as parameters of the `DiscreteChoice` VG function that generates values according to specified probabilities, and would be stored in a separate "parameter table"—there is no notion of "enhanced" tuples.)

Thus, in a ProbDB, the basic relational model is extended to operate on enhanced tuples. Systems such as MayBMS, PrDB, and BayesStore reduce the number of probabilities that need to be stored, using denormalization techniques and exploitation of symmetries in the probability distribution. ProbDBs therefore attach uncertainty to a "deterministic skeleton"—that is, each tuple in the database directly (or in concert with other tuples) produces at most one tuple in each database instance. This restriction seems to preclude some of the most common stochastic models, such as a random walk with a random stopping time. On the other hand, MCDB can trivially implement a host of stochastic models—including VARTA processes [Biller and Nelson 2003], copulas [Nelsen 2006], Indian Buffet Processes [Griffiths and Ghahramani 2005], Dirichlet Processes [Teh et al. 2003], Chinese Restaurant Processes [Blei et al. 2003]—that seem difficult or impossible to handle using ProbDBs. The experimental section of this article offers eleven queries that are all quite realistic; as far as we can tell, the majority of them cannot be run on a ProbDB system. Moreover, MCDB does not extend the standard relational model, but rather puts an uncertainty "wrapper" around the relational model.

Another difference is that, unlike MCDB, most ProbDB systems focus on exact computation of query-result distribution features, such as tuple-inclusion probabilities. Such computation can be performed efficiently under certain conditions on the data and query [Dalvi and Suciu 2007a; Ré and Suciu 2009], but common query types such as aggregation queries can pose a challenge to these systems [Murthy and Widom 2007]. In contrast, the extreme generality of the stochastic models considered by MCDB mandates the use of Monte Carlo methods to infer the effect of the stochastic model on the result of the query, and to determine the accuracy of those inferences.

MCDB's generality does come at a price. If ProbDB systems that rely on analytic methods or special-purpose Monte Carlo algorithms such as Luby-Karp [Ré et al. 2007; Dalvi and Suciu 2007b] can be used to answer a particular query, then we fully expect that they will far outperform MCDB. For example, existing probabilistic databases are undoubtedly far superior for "needle in a haystack" computations, where the goal is to find a key tuple with a one-in-a-million chance of appearing in a result set. To find such a rare tuple, MCDB would need to execute on the order of one million Monte Carlo iterations, which is usually infeasible. Still, MCDB seems to present the easiest mechanism for incorporating general-purpose stochastic modeling into the database in a way that presents a genuine alternative to R, Arena, or Matlab in the sort of application described in the previous section.

We conclude this section by pointing to the other major body of literature related to MCDB, namely, the massive 65-year old literature on Monte Carlo methods. Recent references on the topic include Asmussen and Glynn [2007], Gentle [2003], and Robert and Casella [2004]. MCDB inherits both the strengths—generality, ability to deal with complex high-dimensional probability distributions—and the weaknesses—"needle in a haystack" problems, potentially slow response time—of Monte Carlo technology.

3. A NOTE ON TERMINOLOGY

The individual data-value probability distributions for a random database induce a probability distribution over the set of possible database instances; this set is called the set of *possible worlds* in the literature but we use the terminology *possible DBs* here. The possible-DB distribution then induces a probability distribution over the result of a query over the random database, that is, the *query-result distribution*. MCDB pseudorandomly generates *N* database instances that represent independent samples from the possible-DB distribution. The term *Monte Carlo iteration* refers to the process of generating one database instance and running the query of interest over the instance; thus MCDB executes *N* Monte Carlo iterations. An MCDB relation is *deterministic* if its realization is the same in all possible DBs, otherwise it is *random*. Running a query over a generated database instance produces a *generated query result*, which can be viewed as a random sample from the query-result distribution.

4. SCHEMA SPECIFICATION

We now start to describe MCDB. As mentioned earlier, the system is based on possible-world semantics. Each random relation is specified by a schema, along with a set of VG functions for generating relation instances. The output of a query over a random relation is no longer a single answer, but rather a probability distribution over possible answers. Random relations are specified using an extended version of the SQL `CREATE TABLE` syntax that identifies the VG functions used to generate relation instances, along with the parameters of these functions. In this section we illustrate our specification syntax via a sequence of examples; see Section A of the online appendix available in the ACM Digital Library for a formal BNF specification. We follow Ré et al. [2006] and assume without loss of generality that each random relation *R* can be viewed as a union of blocks of correlated tuples, where tuples in different blocks are independent.

4.1. Schema Syntax: Simple Cases

First consider a very simple setting, in which we wish to specify a table that describes patient systolic blood pressure data, relative to a default of 100 (in units of mm Hg). Suppose that, for privacy reasons, exact values are unavailable, but we know that the average shifted blood pressure for the patients is 10 and that the shifted blood pressure values are normally distributed around this mean, with a standard deviation of 5. Blood pressure values for different patients are assumed independent. Suppose that the aforesaid mean and standard deviation parameters for shifted blood pressure are stored in a single-row table `SPB_PARAM(MEAN, STD)` and that individual patient data are stored in a (deterministic) table `PATIENTS(PID, GENDER)`. Then the random table `SBP_DATA` can be specified as follows.

```
CREATE TABLE SBP_DATA(PID, GENDER, SBP) AS
 FOR EACH p in PATIENTS
  WITH SBP AS Normal (
   SELECT s.MEAN, s.STD
   FROM SPB_PARAM s)
  SELECT p.PID, p.GENDER, b.VALUE
  FROM SBP b
```

A realization of SBP_DATA is generated by looping over the set of patients and using the Normal VG function to generate a row for each patient. These rows are effectively UNIONed to create the realization of SBP_DATA. The FOR EACH clause specifies this outer loop. More generally, every CREATE TABLE specification for a random table has a FOR EACH clause, with each looping iteration resulting in the generation of a block of correlated tuples. The looping variable is tuple-valued, and iterates through the result tuples of either a relation scan (over PATIENTS in our example) or a more general SQL expression.

The standard library VG function Normal generates independent and identically distributed (i.i.d.) pseudorandom samples from a normal distribution, which serve as the uncertain blood pressure values. The mean and variance of this normal distribution is specified in a single-row table that is input as an argument to the Normal function. This single-row table is specified, in turn, as the result of an SQL query—a rather trivial one in this example—over the parameter table SPB_PARAM. The Normal function, like all VG functions, produces a relation as output; in this case, a single-row table having a single attribute, namely, VALUE.

The final SELECT clause assembles the finished row in the realized SBP_DATA table by (trivially) selecting the generated blood pressure from the single-row table created by Normal and appending the appropriate PID and GENDER values. In general, the SELECT clause "glues together" the various attribute values that are generated by one or more VG functions or are retrieved from the outer FOR EACH query and/or from another table. To this end, the SELECT clause may reference the current attribute values of the looping variable, for example, p.PID and p.GENDER.

The current implementation of MCDB produces instances of a random table such as SBP_DATA on demand, during query execution. Other materialization strategies are possible, however; see Section 11.2.

4.2. Parameterizing VG Functions

As a more complicated example, suppose that we wish to create a table of customer data, including the uncertain attributes MONEY, which specifies the annual disposable income of a customer, and LIVES_IN, which specifies the customer's city of residence. The deterministic attributes of the customers are stored in a table

```
CUST_ATTRS(CID, GENDER, REGION).
```

That is, we know the region in which a customer lives but not the precise city. Suppose that, for each region, we associate with each city a probability that a customer lives in that city; thus, the sum of the city probabilities over a region equals 1. These probabilities are contained in a parameter table CITIES(NAME, REGION, PROB). The distribution of the continuous MONEY attribute follows a gamma distribution, which has three parameters: shift, shape, and scale. All customers share the same shift parameter, which is stored in a single-row table MONEY_SHIFT(SHIFT). The scale parameter is the same for all customers in a given region, and these regional scale values are stored in a table MONEY_SCALE(REGION, SCALE). The shape-parameter values vary from customer to customer, and are stored in a table MONEY_SHAPE(CID, SHAPE). The value pairs (MONEY, LIVES_IN) for the different customers are conditionally mutually independent, given the values of REGION and SHAPE for the customers. Similarly, given the REGION value for a customer, the MONEY and LIVES_IN values for that customer are conditionally independent. A specification for the CUST table is then as follows.

```
CREATE TABLE CUST(CID, GENDER, MONEY, LIVES_IN) AS
 FOR EACH d in CUST_ATTRS
  WITH MONEY AS Gamma(
   (SELECT n.SHAPE
    FROM MONEY_SHAPE n
    WHERE n.CID = d.CID)
   (SELECT sc.SCALE
    FROM MONEY_SCALE sc
    WHERE sc.REGION = d.REGION)
   (SELECT SHIFT
    FROM MONEY_SHIFT))
  WITH LIVES_IN AS DiscreteChoice (
    SELECT c.NAME, c.PROB
    FROM CITIES c
    WHERE c.REGION = d.REGION)
  SELECT d.CID, d.GENDER, m.VALUE, l.VALUE
  FROM MONEY m, LIVES_IN l
```

We use the Gamma library function to generate gamma variates; we have specified three single-row, single-attribute tables as input. The DiscreteChoice VG function is a standard library function that takes as input a table of discrete values and selects exactly one value according to the specified probability distribution.

Note that by modifying MONEY_SHAPE, MONEY_SCALE, and MONEY_SHIFT, we automatically alter the definition of CUST, allowing what-if analyses to investigate the sensitivity of query results to probabilistic assumptions and the impact of different scenarios (e.g., an income-tax change may affect disposable income). Another type of what-if analysis that we can easily perform is to simply replace the Gamma or DiscreteChoice functions in the definition of CUST with alternative VG functions.

4.3. Capturing ProbDB Functionality

In this and the following section, we indicate how MCDB can capture the functionality of ProbDBs (though perhaps not always as efficiently; see Section 2). As a variant of the preceding example, suppose that associated with each customer is a set of possible cities of residence, along with a probability for each city; this information is stored in a table CITIES(CID, NAME, PROB). We then change the definition of LIVES_IN to the following.

```
WITH LIVES_IN AS DiscreteChoice (
  SELECT c.NAME, c.PROB
  FROM CITIES c
  WHERE c.CID = d.CID)
```

Thus, MCDB can capture attribute-value uncertainty [Agrawal et al. 2006; Antova et al. 2007; Gupta and Sarawagi 2006].

Tuple-inclusion uncertainty as in Dalvi and Suciu [2007b] can also be represented within MCDB. Consider a variant of the example of Section 4.2 in which the CUST_ATTRS table has an additional attribute INCL_PROB which indicates the probability that the customer truly belongs in the CUST table. To represent inclusion uncertainty, we use the library VG function Bernoulli, which takes as input a single-row table with a single attribute PROB and generates a single-row, single-attribute output table, where the attribute VALUE equals true with probability p specified by PROB and equals false with probability $1 - p$. Augment the original CUST table specification with the clause

```
WITH IN_TABLE AS Bernoulli (VALUES(d.INCL_PROB))
```

where, as in standard SQL, the VALUES function produces a single-row table whose entries correspond to the input arguments. Also modify the select clause as follows.

```
  SELECT d.CID, d.GENDER, m.VALUE, l.VALUE
  FROM MONEY m, LIVES_IN l, IN_TABLE i
  WHERE i.VALUE = true
```

4.4. Structural Uncertainty

“Structural” uncertainty [Getoor and Taskar 2007], that is, fuzzy queries, can also be captured within the MCDB framework. For example, suppose that a table LOCATION(LID, NAME, CITY) describes customer locations, and a table SALES(SID, NAME, AMOUNT) holds transaction records for these customers. We would like to compute sales by city, and so need to join the tables LOCATION and SALES. We need to use a fuzzy similarity join because a name in LOCATION and name in SALES that refer to the same entity may not be identical, because of spelling errors, different abbreviations, and so forth. Suppose that we have a similarity function Sim that takes two strings as input, and returns a number between 0 and 1 that can be interpreted as the probability that the two input strings refer to the same entity. Then we define the following random relation.

```
CREATE TABLE LS_JOIN (LID, SID) AS
 FOR EACH t IN (
    SELECT l.LID, l.NAME AS NAME1,
           s.SID, s.NAME AS NAME2
    FROM LOCATIONS l, SALES s)
  WITH JOINS AS Bernoulli (
   VALUES(Sim(t.NAME1, t.NAME2)))
  SELECT t.LID, t.SID
  FROM JOINS j
  WHERE j.VALUE = true
```

Here Bernoulli is defined as before. The desired sales query is then as follows.

```
SELECT l.CITY, SUM(s.AMOUNT)
FROM LOCATION l, SALES s, LS_JOIN j
WHERE l.LID = j.LID AND s.SID = j.SID
GROUP BY l.CITY
```

Unlike the traditional approach, in which all tuples that are “sufficiently” similar are joined, repeated Monte Carlo execution of this query in MCDB yields information not only about the “most likely” answer to the query, but about the entire distribution of sales amounts for each city. We can then assess risk, such as the probability that sales for a given city lie below some critical threshold.

4.5. Correlated Attributes and Tuples

In general, a VG function takes zero or more parameter tables as input, and returns a table containing pseudorandomly generated values. Until now, we have implicitly focused on VG functions that return a one-row, one-column table, that is, a single generated value. VG functions that return tables containing multiple generated values can capture statistical correlations within or between tuples.

Correlated attributes within a tuple are easily handled by using VG functions whose output table has multiple columns. Consider the case where a customer’s income and city of residence are correlated.

```
CREATE TABLE CUST(CID, GENDER, MONEY, LIVES_IN) AS
 FOR EACH d in CUST_ATTRS
  WITH MLI AS MyJointDistribution (...)
  SELECT d.CID, d.GENDER, MLI.VALUE1, MLI.VALUE2
  FROM MLI
```

The user-defined VG function MyJointDistribution outputs a single-row table with two (correlated) attributes VALUE1 and VALUE2 that correspond to the generated values of MONEY and LIVES_IN.

Correlations between tuples can be handled in an analogous manner. Suppose, for example, that we have readings from a collection of temperature sensors. Because of uncertainty in the sensor measurements, we view each reading as the mean of a normal probability distribution. We assume that the sensors are divided into groups, where sensors in the same group are located close together, so that their readings are correlated, and thus the group forms a multivariate normal distribution. The table `S_PARAMS(ID, LAT, LONG, GID)` contains the sensor ID (a primary key), the latitude and longitude of the sensor, and the group ID. The means corresponding to the given "readings" are stored in a parameter table `MEANS(ID, MEAN)`, and the correlation structure is specified by a covariance matrix whose entries are stored in a parameter table `COVARS(ID1, ID2, COVAR)`. The desired random table `SENSORS` is then specified as follows.

```
CREATE TABLE SENSORS(ID, LAT, LONG, TEMP) AS
 FOR EACH g IN (SELECT DISTINCT GID FROM S_PARAMS)
  WITH TEMP AS MDNormal(
   (SELECT m.ID, m.MEAN
    FROM MEANS m, SENSOR_PARAMS ss
    WHERE m.ID = ss.ID AND ss.GID = g.GID)
   (SELECT c.ID1, c.ID2, c.COVAR
    FROM COVARS c, SENSOR_PARAMS ss
    WHERE c.ID1 = ss.ID AND ss.GID = g.GID))
  SELECT s.ID, s.LAT, s.LONG, t.VALUE
  FROM SENSOR_PARAMS s, TEMP t
  WHERE s.ID = t.ID
```

The subquery in the `FOR EACH` clause creates a single-attribute relation containing the unique group IDs, so that the looping variable `g` iterates over the sensor groups. The `MDNormal` function is invoked once per group, that is, once per distinct value of `g`. For each group, the function returns a multirow table having one row per group member. This table has two attributes: `ID`, which specifies the identifier for each sensor in the group, and `VALUE`, which specifies the corresponding generated temperature. The join that is specified in the final `SELECT` clause serves to append the appropriate latitude and longitude to each tuple produced by `MDNormal`, thereby creating a set of completed rows.

In general, by judicious use of multiple VG functions that each return multiple correlated values, the MCDB user can model a broad variety of correlation patterns within and between tuples.

5. SPECIFYING VG FUNCTIONS

A user of MCDB can take advantage of a standard library of VG functions, such as `Normal()` or `Poisson()`, or can implement VG functions that are linked to MCDB at query-processing time. The latter class of customized VG functions is specified in a manner similar to the specification of UDFs in ordinary database systems.

Each VG function is implemented as a C++ class having four `public` methods: `Initialize()`, `TakeParams()`, `OutputVals()`, and `Finalize()`. For each VG function referenced in a `CREATE TABLE` statement, the following sequence of events is initiated for each tuple in the `FOR EACH` clause.

First, MCDB calls the `Initialize()` method with the seed that the VG function will use for pseudorandom number generation.[2] This invocation instructs the VG function to set up any data structures that will be required.

[2] A uniform pseudorandom number generator deterministically and recursively computes a sequence of *seed* values—n-bit integers, where the value of n depends on the specific generator—which are then converted to floating-point numbers in the range $[0, 1]$. Although this process is deterministic, the floating-point numbers produced by a well-designed generator will be statistically indistinguishable from a sequence of "truly" i.i.d. uniform random numbers. See Fishman [1996] for an introductory discussion. The uniform pseudorandom

Next, MCDB executes the queries that specify the input parameter tables to the VG function. The result of the query execution is made available to the VG function in the form of a sequence of arrays called *parameter vectors*. The parameter vectors are fed into the VG function via a sequence of calls to `TakeParams()`, with one parameter vector at each call. See Section C of the online appendix available in the ACM Digital Library for details.

After parameterizing the VG function, MCDB then executes the first Monte Carlo iteration by repeatedly calling `OutputVals()` to produce the rows of the VG function's output table, with one row returned per call. MCDB knows that the last output row has been generated when `OutputVals()` returns a `NULL` result. Such a sequence of calls to `OutputVals()` can then be repeated to generate the second Monte Carlo replicate, and so forth. When all of the required Monte Carlo replicates have been generated, MCDB calls the VG function's `Finalize()` method, which deletes any internal data structures. Section B of the online appendix contains an example of how a simple VG function can be implemented.

6. QUERY PROCESSING IN MCDB

In this section we describe the basic query-processing ideas underlying our prototype implementation. Subsequent sections contain further details.

6.1. A Naive Implementation

Conceptually, the MCDB query processing engine evaluates a query Q over many different database instances, and then uses the various result sets to estimate the appearance probability for each result tuple. These estimated probabilities can then be used to compute other statistical quantities of interest; see Section 12. It is easy to imagine a simple method for implementing this process. Given a query Q over a set of deterministic and random relations, the following three steps would be repeated N times, where N is the number of Monte Carlo iterations specified.

(1) Generate an instance of each random relation as specified by the various `CREATE TABLE` statements.
(2) Once an entire instance of the database has been materialized, compile, optimize, and execute Q in the classical manner.
(3) Append every tuple in Q's answer set with a number identifying the current Monte Carlo iteration.

Once N different query results have been generated, all of the output tuples are then merged into a single file, sorted, and scanned to determine the number of query results in which each tuple appears.

Unfortunately, although this basic scheme is quite simple, it is likely to have dismal performance in practice. The obvious problem is that each individual database instance may be very large, perhaps terabytes in size, and N is likely to be somewhere from 10 to 1000. Thus, this relatively naive implementation is impractical, and so MCDB uses a very different strategy.

6.2. Overview of Query Processing in MCDB

The key ideas behind MCDB query processing are as follows.

MCDB executes the query exactly once, regardless of N. In MCDB, Q is evaluated only *once*, whatever the value of N supplied by the user. Each "database tuple" that

numbers can then be transformed into pseudorandom numbers having the desired final distribution [Devroye 1986].

is processed by MCDB is actually an array or "bundle" of tuples, where $t[i]$ for tuple bundle t denotes the value of t in the ith Monte Carlo iteration.

The potential performance benefit of the "tuple bundle" approach is that relational operations may efficiently operate in batch across all N Monte Carlo iterations that are encoded in a single tuple bundle. For example, if $t[i].att$ equals some constant c for all i, then the relational selection operation $\sigma_{att=7}$ can be applied to $t[i]$ for all possible values of i via a single comparison with the value c. Thus, bundling can yield a N-fold reduction in the number of tuples that must be moved through the system and processed. Moreover, if many bundles are filtered out by selections as in the preceding example, and if the filtering occurs early in the query plan, then much of the Monte Carlo work can be avoided.

MCDB delays random attribute materialization as long as possible. The obvious cost associated with storing all of the N generated values for an attribute in a tuple bundle is that the resulting bundle can be very large for large N. If $N = 1000$ then storing all values for a single random character string can easily require 100KB per tuple bundle. MCDB alleviates this problem by materializing attribute values for a tuple as late as possible during query execution, typically right before random attributes are used by some relational operation.

In MCDB, values for random attributes are reproducible. After an attribute value corresponding to a given Monte Carlo iteration has been materialized (as described before) and processed by a relational operator, MCDB permits this value to be discarded and then later rematerialized if it is needed by a subsequent operator. To ensure that the same value is generated each time, so that the query result is consistent, MCDB ensures that each tuple carries the pseudorandom number seeds that it supplies to the VG functions. Supplying the same seed to a given VG function at every invocation produces identical generated attribute values. One can view the seed value as being a highly compressed representation of the random attribute values in the tuple bundle.

7. TUPLE BUNDLES IN DETAIL

A tuple bundle t with schema $\mathcal{S}$ is, logically speaking, simply an array of N tuples—all having schema $\mathcal{S}$—where N is the number of Monte Carlo iterations; we denote by $t[i]$ the ith tuple in the array. Tuple bundles are manipulated using the new operators described in Section 8 and the modified versions of classical relational operators described in Section 9.

Individual tuples can be bundled together across database instances in many ways. The only requirement on a set of tuple bundles $t_1, t_2, \ldots, t_k$ is that, for each $i \in [1..N]$, the set $r_i = \bigcup_j t_j[i]$ corresponds precisely to the ith generated instance of R. For storage and processing efficiency, MCDB tries to bundle tuples so as to maximize the number of "constant" attributes. An attribute att is constant in a tuple bundle t if $t[i].att = c$ for some fixed value c and $i = 1, 2, \ldots, N$. Since constant attributes do not vary across Monte Carlo iterations, they can be stored in compressed form as a single value. In the blood pressure example of Section 4.1, the natural approach is to have one tuple bundle for each patient, since then the patient ID is a constant attribute. Attributes that are supplied directly from deterministic relations are constant. MCDB also allows the implementor of a VG function to specify attributes as constant as a hint to the system. Then, when generating Monte Carlo replicates of a random relation, MCDB creates one tuple bundle for every distinct combination of constant-attribute values encountered. As mentioned previously, MCDB often stores values for nonconstant attributes in a highly compressed form by storing only the seed used to pseudorandomly generate the values, rather than an actual array of generated values.

A tuple bundle t in MCDB may have a special random attribute called the *isPresent* attribute. The value of this attribute in the ith database instance is denoted by $t[i].isPres$. The value of $t[i].isPres$ equals `true` if and only if the tuple bundle actually has a constituent tuple that appears in the ith database instance. If the *isPresent* attribute is not explicitly represented in a particular tuple bundle, then $t[i].isPres$ is assumed to be `true` for all i, so that t appears in every database instance.

isPresent is not created via an invocation of a VG function. Rather, it may result from a standard relational operation that happens to reference an attribute created by a VG function. For instance, consider a random attribute `gender` that takes the value `male` or `female`, and the relational selection operation σ_B where B is the predicate "`gender` $=$ `female`". If, in the ith database instance, $t[i]$.`gender` $=$ `male`, then $t[i].isPres$ will necessarily be set to `false` after application of σ_B to t because σ_B removes t from that particular database instance. In MCDB the *isPresent* attribute is physically implemented as an array of N bits within the tuple bundle, where the ith bit corresponds to $t[i].isPres$.

8. NEW OPERATIONS IN MCDB

Under the hood, MCDB's query processing engine looks quite similar to a classical relational query processing engine. The primary differences are that: (1) MCDB implements a few additional operations, and (2) the implementations of most of the classic relational operations must be modified slightly to handle the fact that tuple bundles, and not tuples, move through the query plan. We begin by describing in some detail the operations unique to MCDB.

8.1. The Seed Operator

Consider a given random relation R whose definition involves one or more VG functions. For every tuple created by R's `FOR EACH` statement, the `Seed` operator appends one or more integers to the tuple, with one integer per VG function. Each such integer is unique to the (tuple, VG function) pair, and serves as a pseudorandom seed when using the VG function to instantiate random-attribute values for the various possible DBs. The `Seed` operator thus transforms an ordinary tuple into a compressed tuple bundle. The seeds must be assigned and used carefully in order to ensure statistically valid results; see Section 11.1 for implementation details.

8.2. The Instantiate Operator

The `Instantiate` operator is responsible for actually producing the random data that is used by MCDB; in effect, `Instantiate` implements the stochastic `CREATE TABLE` statement. This operator has two inputs. First, it accepts a stream of seeded tuples corresponding to the `FOR EACH` clause in the random relation's `CREATE TABLE` statement. Second, it accepts zero or more streams of tuples, each of which corresponds to one of the "inner" queries used to parameterize the VG function that produces the stochastic data, for example, the two queries that produce the mean and covariance parameters used in the `MDNormal` VG function in the sensor example of Section 4.5. `Instantiate` then uses the seeds, along with the VG function parameters, to produce a stream of expanded tuple bundles. Space precludes us from discussing the inner workings of `Instantiate` in detail, but such a discussion can be found in the online appendix (Section D) available in the ACM Digital Library; see also Section 11.3.

8.3. The Split Operator

One potential problem with the "tuple bundle" approach is that it becomes impossible to order tuple bundles with respect to a nonconstant attribute, as is needed for a variety of relational operators. MCDB therefore implements the `Split` operator, which is used—as described in subsequent sections—within the sort-merge join operator, the duplicate elimination operator, the multiset union, intersection, and difference operators, and the new `Inference` operator.

The `Split` operator takes as input a tuple bundle, together with a set of attributes *Atts*. `Split` then splits the tuple bundle into multiple tuple bundles, such that, for each output bundle, each of the attributes in *Atts* is now a constant attribute. Moreover, the constituent tuples for each output bundle t are marked as nonexistent (that is, $t[i].isPres =$ `false`) for those Monte Carlo iterations in which t's particular set of *Atts* values is not observed. For example, consider a tuple bundle t with schema `(fname, lname, age)` where attributes `fname = Jane` and `lname = Smith` are constant, and attribute `age` is nonconstant. Specifically, suppose that there are four Monte Carlo iterations and that $t[i]$`.age = 20` for $i = 1, 3$ and $t[i]$`.age = 21` for $i = 2, 4$. We can compactly represent this tuple bundle as

$$t = \texttt{(Jane, Smith, (20,21,20,21),(T,T,T,T))},$$

where the last nested vector contains the *isPresent* values, and indicates that Jane Smith appeared in all four Monte Carlo iterations (though with varying ages). An application of the `Split` operation to t with $Atts = \{\texttt{age}\}$ yields two tuple bundles.

$$t_1 = \texttt{(Jane, Smith, 20, (T, F, T, F))}$$
$$t_2 = \texttt{(Jane, Smith, 21, (F, T, F, T))}.$$

Thus, the nondeterminism in `age` has been transferred to the *isPresent* attribute.

8.4. The Inference Operator

The final new operator in MCDB is the `Inference` operator. The output from this operator is a set of distinct, ordinary tuples. Each such tuple t' is annotated with a value f that denotes the fraction of the Monte Carlo iterations for which t' appears at least once in the query result. Note that f estimates the true probability that t' will appear in a realization of the query result.

MCDB implements the `Inference` operator as follows. Assume that the input query returns a set of tuple bundles with exactly the set of attributes *Atts* (not counting the *isPresent* attribute). Then:

(1) MCDB runs the `Split` operation on each tuple bundle in Q using *Atts* as the attribute-set argument. This ensures that each resulting tuple bundle has all of its nondeterminism "moved" to the *isPresent* attribute.
(2) Next, MCDB runs the duplicate removal operation (see the next section for a description).
(3) Finally, for each resulting tuple bundle, `Inference` counts the number n of i values for which $t[i].isPres =$ `true`. The operator then outputs a tuple with attribute value $t[\cdot].att$ for each $att \in Atts$, together with the relative frequency $f = n/N$. For example, if bundle t_1 in Section 8.3 results from steps (1) and (2) given before, then the `Inference` operator outputs the tuple `(Jane, Smith, 20, 0.5)`.

9. STANDARD RELATIONAL OPS

In addition to the new operations described earlier, MCDB implements versions of the standard relational operators that are modified to handle tuple bundles.

9.1. Relational Selection

Given a boolean relational selection predicate B and a tuple bundle t, for each i, MCDB sets $t[i].isPres \leftarrow B(t[i]) \wedge t[i].isPres$. In the case where $t.isPres$ has not been materialized and stored with t, then $t[i].isPres$ is assumed to equal `true` for all i prior to the selection, and $t[i].isPres$ is set to $B(t[i])$.

If, after application of B to t, $t[i].isPres = \texttt{false}$ for all i, then t is rejected by the selection predicate and t is not output at all by $\sigma_B(t)$. If B refers only to constant attributes, then the time to perform the selection is independent of the number N of Monte Carlo iterations.

9.2. Projection

Projection in MCDB is nearly identical to projection in a classical system, with a few additional considerations. If a nonconstant attribute is projected away, the entire array of values for that attribute is removed. Also, so that an attribute generated by a VG function can be regenerated, projection of an attribute does not remove the seed for that attribute unless explicitly specified in the query plan.

9.3. Cartesian Product and Join

The Cartesian product operation ($\times$) in MCDB is also similar to the classical relational case. Assume we are given two sets of tuple bundles R and S. For $r \in R$ and $s \in S$, define $t = r \oplus s$ to be the unique tuple bundle such that:

(1) $t[i] = r[i] \bullet s[i]$ for all i, where $\bullet$ denotes tuple concatenation, but excluding the elements $r[i].isPres$ and $s[i].isPres$.
(2) $t[i].isPres = r[i].isPres \wedge s[i].isPres$.

Then the output of the $\times$ operation comprises all such t.

The join operation ($\bowtie$) with an arbitrary boolean join predicate B is logically equivalent to a $\times$ operation as before, followed by an application of the (modified) relational selection operation σ_B. In practice, B most often contains an equality check across the two input relations (i.e., an equijoin). An equijoin over constant attributes is implemented in MCDB using a sort-merge algorithm. An equijoin over nonconstant attributes is implemented by first applying the `Split` operation to force all of the join attributes to be constant, and then using a sort-merge algorithm.

9.4. Duplicate Removal

To execute the duplicate-removal operation, MCDB first executes the `Split` operation, if necessary, to ensure that *isPresent* is the only nonconstant attribute in the input tuple bundles. The bundles are then lexicographically sorted according to their attribute values (excluding *isPresent*). This sort operation effectively partitions the bundles into groups such that two bundles are in the same group if and only if they have identical attribute values. For each such group T, exactly one result tuple t is output. The attribute values of t are the common ones for the group, and

$$t[i].isPres = \bigvee_{t' \in T} t'[i].isPres$$

for each i.

9.5. Aggregation

To sum a set of tuple bundles T over an attribute *att*, MCDB creates a result tuple bundle t with a single attribute called *agg* and sets $t[i].agg = \sum_{t' \in T} I(t'[i].isPres) \times t'[i].att$. In this expression, I is the indicator function returning 1 if $t'[i].isPres = \texttt{true}$ and 0 otherwise. Other aggregation functions are implemented similarly. Note that the actual result of an aggregate operation never has an *isPres* attribute, because an aggregate query *always* has a result, even if it is given empty input. In this latter case, $t[i].agg = \texttt{NULL}$.

9.6. Multiset Union, Intersection, and Difference

To implement the operations $R \cup S$, $R \cap S$, and $R \setminus S$ over two random relations R and S, MCDB first executes a `Split` operation, if necessary, to ensure that *isPresent* is the only nonconstant attribute in R and S. Tuple bundles in R are partitioned into groups as in Section 9.4, so that each group corresponds to a distinct tuple value (ignoring *isPresent*). For a group G of bundles corresponding to a distinct tuple value t, the *isPresent* bit vectors for the tuples are treated as numerical 0/1 vectors and summed, to create a count vector $c_{t,R}$ with $c_{t,R}[i] = \sum_{t' \in G} t'[i].\mathit{isPresent}$. Thus $c_{t,R}[i]$ is the number of times that t appears in the ith Monte Carlo iteration. The $c_{R,t}$ vector is then attached to t. This process results in a set of pairs (t, c_R) for R, with all t values distinct. The same processing is performed for S. The (t, c_R) pairs for R are then joined to the (t, c_S) pairs for S on the t attributes, to form tuples of the form (t, c_R, c_S). For each (t, c_R, c_S) tuple, c_R and c_S are combined into a count vector c, with $c[i]$ equal to $c_R[i] + c_S[i]$, $\min(c_R[i], c_S[i])$, and $\max(c_R[i] - c_S[i], 0)$, for the cases of multiset union, intersection, and difference, respectively. Then t is replicated to create a total of m copies, where $m = \max_i c[i]$. Each copy of t is then augmented with an *isPresent* vector, where the bit values for each vector are chosen so that the vector sum is equal to c. For example, if there are $N = 4$ Monte Carlo iterations and $c = (3, 0, 1, 3)$, then $m = 3$ and one possibility for the three *isPresent* vectors is 1011, 1001, and 1001. For the special case of ordinary set operations, c_R, c_S, and c vectors can be computed using appropriate bitwise ANDing and ORing operations.

10. EXAMPLE QUERY IMPLEMENTATION

We can make all of this a bit more concrete by tracing the evaluation of an example query from start to finish in MCDB. Consider two random relations `SALE(name, amt, when)` and `CUST(name, loc)`, where `when` and `loc` are the random attributes. Suppose that we run the following SQL query, which computes the total revenue resulting from U.S.-based "frequent" customers who made at least two purchases within 30 days of one another.

```
WITH FreqCust(name) AS (
  SELECT DISTINCT c.name
    FROM SALE s1, SALE s2, CUST c
    WHERE c.loc = 'US' AND c.name = s1.name AND c.name = s2.name
          AND s1.when <> s2.when AND s2.when - s1.when <= 30)
SELECT SUM(amt)
FROM SALE, FreqCust
 WHERE SALE.name = FreqCust.name
```

Figure 3 gives a plan and example execution for this query. For brevity, we do not depict the evaluation of any of the subqueries that provide parameters to the various VG functions. Thus, in this figure, we only show a single input into each `Instantiate` operator: the relation that supplies all of the `FOR EACH` tuples in the corresponding `CREATE TABLE` statement. Moreover, we do not depict as separate operations the majority of the relational projections; in the figure, attributes are simply dropped when they are no longer needed. The only exception is the one projection operation depicted at the bottom of the figure. This projection is made explicit so that, for clarity, we can show the result of the previous relational selection.

The query is evaluated as follows. First, the `CUST` tuples are seeded and then `Instantiate` is used to produce values of the `loc` attribute. This results in a set of tuple bundles, each having an array of possible location values. The result is pushed through a relational selection that filters out all customers not from the United States.

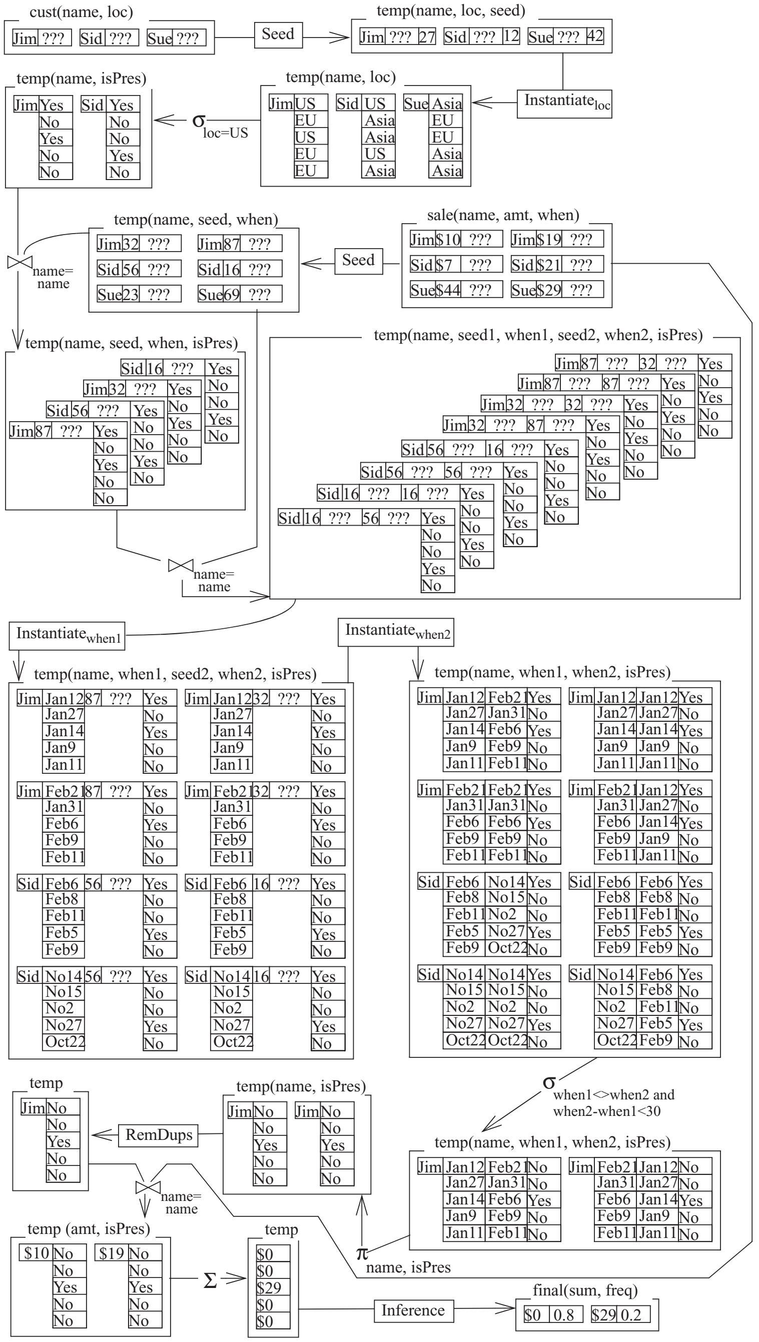
cust(name, loc)
Jim ??? | Sid ??? | Sue ???
Seed
temp(name, loc, seed)
Jim ??? 27 | Sid ??? 12 | Sue ??? 42
Instantiate$_{loc}$
temp(name, loc)
Jim US EU US EU EU | Sid US Asia Asia US Asia | Sue Asia EU EU Asia Asia
σ loc=US
temp(name, isPres)
Jim Yes No Yes No No | Sid Yes No No Yes No
temp(name, seed, when)
Jim 32 ??? | Jim 87 ???
Sid 56 ??? | Sid 16 ???
Sue 23 ??? | Sue 69 ???
Seed
sale(name, amt, when)
Jim $10 ??? | Jim $19 ???
Sid $7 ??? | Sid $21 ???
Sue $44 ??? | Sue $29 ???
name= name
temp(name, seed, when, isPres)
Sid 16 ??? Yes No No Yes No
Jim 32 ??? Yes No Yes No No
Sid 56 ??? Yes No No Yes No
Jim 87 ??? Yes No Yes No No
temp(name, seed1, when1, seed2, when2, isPres)
Jim 87 ??? 32 ??? Yes No Yes No No
Jim 87 ??? 87 ??? Yes No Yes No No
Jim 32 ??? 32 ??? Yes No Yes No No
Jim 32 ??? 87 ??? Yes No Yes No No
Sid 56 ??? 16 ??? Yes No No Yes No
Sid 56 ??? 56 ??? Yes No No Yes No
Sid 16 ??? 16 ??? Yes No No Yes No
Sid 16 ??? 56 ??? Yes No No Yes No
name= name
Instantiate$_{when1}$
Instantiate$_{when2}$
temp(name, when1, seed2, when2, isPres)
Jim Jan12 Jan27 Jan14 Jan9 Jan11 87 ??? Yes No Yes No No | Jim Jan12 Jan27 Jan14 Jan9 Jan11 32 ??? Yes No Yes No No
Jim Feb21 Jan31 Feb6 Feb9 Feb11 87 ??? Yes No Yes No No | Jim Feb21 Jan31 Feb6 Feb9 Feb11 32 ??? Yes No Yes No No
Sid Feb6 Feb8 Feb11 Feb5 Feb9 56 ??? Yes No No Yes No | Sid Feb6 Feb8 Feb11 Feb5 Feb9 16 ??? Yes No No Yes No
Sid No14 No15 No2 No27 Oct22 56 ??? Yes No No Yes No | Sid No14 No15 No2 No27 Oct22 16 ??? Yes No No Yes No
temp(name, when1, when2, isPres)
Jim Jan12 Feb21 Yes / Jan27 Jan31 No / Jan14 Feb6 Yes / Jan9 Feb9 No / Jan11 Feb11 No | Jim Jan12 Jan12 Yes / Jan27 Jan27 No / Jan14 Jan14 Yes / Jan9 Jan9 No / Jan11 Jan11 No
Jim Feb21 Feb21 Yes / Jan31 Jan31 No / Feb6 Feb6 Yes / Feb9 Feb9 No / Feb11 Feb11 No | Jim Feb21 Jan12 Yes / Jan31 Jan27 No / Feb6 Jan14 Yes / Feb9 Jan9 No / Feb11 Jan11 No
Sid Feb6 No14 Yes / Feb8 No15 No / Feb11 No2 No / Feb5 No27 Yes / Feb9 Oct22 No | Sid Feb6 Feb6 Yes / Feb8 Feb8 No / Feb11 Feb11 No / Feb5 Feb5 Yes / Feb9 Feb9 No
Sid No14 No14 Yes / No15 No15 No / No2 No2 No / No27 No27 Yes / Oct22 Oct22 No | Sid No14 Feb6 Yes / No15 Feb8 No / No2 Feb11 No / No27 Feb5 Yes / Oct22 Feb9 No
σ when1<>when2 and when2-when1<30
temp(name, when1, when2, isPres)
Jim Jan12 Feb21 No / Jan27 Jan31 No / Jan14 Feb6 Yes / Jan9 Feb9 No / Jan11 Feb11 No | Jim Feb21 Jan12 No / Jan31 Jan27 No / Feb6 Jan14 Yes / Feb9 Jan9 No / Feb11 Jan11 No
π name, isPres
temp(name, isPres)
Jim No No Yes No No | Jim No No Yes No No
RemDups
temp
Jim No No Yes No No
name= name
temp (amt, isPres)
$10 No No Yes No No | $19 No No Yes No No
Σ
temp
$0 $0 $29 $0 $0
Inference
final(sum, freq)
$0 0.8 | $29 0.2

Fig. 3. Evaluation of a multitable query in MCDB.

This operation kills Sue's tuple bundle (since there is no database instance where she is from the United States) and sets the *isPres* attribute in the other two bundles. The ith *isPres* bit for a customer is set to "yes" (i.e., to 1) if and only if the customer is from the United States in the ith Monte Carlo iteration.

The SALE tuples are seeded as well, and then the result is joined twice with those tuple bundles from CUST that survived the selection on the loc attribute. (Two joins are needed because of the two instances of SALE in the subquery of the original SQL query.) This results in an eight-tuple-bundle temporary relation, with four tuple bundles for Jim and four for Sid. At this point, it is necessary to check whether either of these customers had two sales within 30 days of each other. So, the temporary relation is piped through a sequence of two Instantiate operators; one that actually materializes the first when, and one that materializes the second. A selection then eliminates all tuples that do not satisfy the 30-day criterion in any Monte Carlo iteration. Only the two tuples associated with Jim survive this selection (because the criterion is satisfied in the third Monte Carlo iteration). A duplicate-removal operation then results in a single Jim tuple. This tuple is then joined with the original SALE relation to obtain all of Jim's sales. The aggregate operation finds that there is only one Monte Carlo iteration where the result is nonzero. Finally, the result of the aggregation is piped into the Inference operation, which produces the final, empirical distribution that is the result of the query.

11. ADDITIONAL DETAILS

We discuss in more detail pseudorandom number generation, materialization of database instances, and parallelization of the instantiate operation.

11.1. Pseudorandom Variate Generation

The two main tasks in pseudorandom number generation comprise seeding the tuple bundles with the Seed operator, and then using the seeds within a VG function when the bundle is instantiated. As mentioned previously, seeds must be assigned and exploited carefully; this section discusses the key challenges and implementation solutions. After briefly reviewing some basic facts about PseudoRandom Number Generators (PRNGs), we describe the seeding operation for the case of a single VG function; the extension to multiple VG functions is straightforward.

In general, a PNRG is initiated with a starting seed s_0, and then generates a sequence of seeds $s_1, s_2, \ldots$ by using a deterministic recursion of the form $s_{i+1} = T(s_i)$. At each step, the generator typically uses a second transformation to create a 32-bit integer $r_i = T'(s_i) \in \{1, 2, \ldots, 2^{32} - 1\}$, which is further transformed to a pseudorandom uniform number u_i on $[0, 1]$ via normalization: $u_i = r_i/2^{32}$. The transformations T and T' depend on the specific generator, as do the number of bits in each s_i (typically a multiple of 16). The sequence of seeds produced by a PRNG eventually loops back on itself—that is, $s_N = s_0$ for some number N with $s_i \neq s_j$ for $0 \leq i < j < N$—thus forming a *cycle* of seeds. The number N is called the *cycle length* of the generator.

Clearly, we want to generate pseudorandom numbers that are of good quality from a statistical perspective, and this is achieved in MCDB by using well-tested and widely accepted Pseudorandom Number Generators (PRNGs), as discussed shortly. A more challenging problem is to minimize the chance that the sequences of seeds "consumed" by any two tuple bundles during their instantiations overlap, because such overlaps can induce unintended statistical correlations between the random values that are generated for the two bundles. Further complicating this problem is the fact that the number of seeds consumed cannot always be determined in advance. For example, a VG function might use an acceptance-rejection algorithm [Devroye 1986, Section II.3] to obtain a pseudorandom sample from a nonuniform distribution; such an algorithm consumes a varying, unpredictable number of input uniform pseudorandom numbers (and hence seeds) at each execution. Another challenge is to minimize the length of the seed that needs to be carried with each tuple; unfortunately, the shorter the seed, the shorter the period of the associated PNRG, and the greater the likelihood of seed-sequence overlaps between bundles.

Our initial seeding implementation addresses the foregoing issues using a simple approach that involves two PNRGs. The first PRNG, denoted G_1, is the LRAND48 generator from the UNIX standard library. For this generator, a seed comprises 48 bits, stored as an array of three 16-bit short integers, and the basic transformation is $T(s) = (as + b) \bmod 2^{48}$, where $a = 25214903917$ and $b = 11$. The generator returns the most significant 32 bits of the seed to the user and does not normalize to the unit interval: in our previous notation, $T'(s) = s/2^{16}$. The second PRNG, denoted G_2, is the WELL512 generator of Panneton et al. [2006], which has a seed length of 512 bits, represented as an array of sixteen 32-bit integers. The basic transformation is of the form $T(s) = As \bmod 2$ for an appropriate binary matrix A, and the secondary transformation $T'(s)$ performs certain bit operations on s and then returns the 32 most significant bits, which are then normalized to form a uniform pseudorandom number.

When seeding a set of tuples, the first tuple is seeded with a fixed 48-bit seed s_0 that specifies a starting point for the G_1 cycle. The next tuple bundle is seeded with s_{16}, which is computed by repeated iteration of G_1's transformation operator T, then the next bundle is seeded with s_{32}, and so forth. Thus each tuple is augmented with 48 bits of seeding data. When instantiating a tuple bundle that has been seeded with the 48-bit seed s_i (where $i = 16m$ for some nonnegative integer m), generator G_1 is first used to materialize the output sequence of 32-bit integers $r_i, r_{i+1}, ..., r_{i+15}$. These integers are then concatenated to form a 512-bit seed s'_0, which is then used to initialize generator G_2. This latter generator is then invoked by the VG function during instantiation. This approach does not guarantee that the G_2 seed sequences for distinct bundles will not overlap. Observe, however, that the sequence used by any given tuple bundle starts at essentially a random position on the G_2 cycle. The G_2 cycle length is 2^{512} ($> 10^{154}$), so that, even with millions of bundles consuming millions of seeds apiece, the likelihood of an overlap is vanishingly small. There are many possible optimizations to this basic approach. For example, we could seed successive tuples with G_1 seeds that are 11 positions apart and then, at instantiation time, run G_1 to create 48-bit seeds $s_i, s_{i+1}, \ldots, s_{i+10}$, for a total of $11 \times 48 = 528$ bits, of which the first 512 can be used to initialize G_2; this requires, however, that we access and manipulate the internal state of G_1, whereas our proposed approach is simpler in that it lets G_1 be treated as a black box that produces 32-bit integers. More sophisticated approaches to seeding are also possible, by adapting or extending results from the parallel simulation literature; see, for example, Srinivasan et al. [1997], Tan [2002], and references therein.

11.2. Materialization of Random Relations

One could imagine two choices for when the system actually materializes a random relation's data, that is, the set of instantiated tuple bundles for the relation. The data could be materialized (via execution of the `CREATE TABLE` statement), written to disk, and then queried subsequently, or the data could be materialized on demand at query time.

Each approach has potential advantages. Materializing data at query time ensures that the parameterization of the relation's VG functions is current. Indeed, if the tuple bundles are prematerialized at table-definition time and then the data changes, or the definition of the random relation changes, or the required number of Monte Carlo iterations changes, much or all of the materialization work will be wasted. If the relation has many tuples or if the number of Monte Carlo iterations is large (so that each instantiated tuple bundle is large), then materialization at query time avoids the need to store and retrieve massive amounts of data.

On the other hand, materializing the data prior to query execution can potentially save CPU resources, especially if the VG function is expensive and the query plan would require a given set of tuple bundles to be instantiated multiple times. There would also be a potential for amortizing VG function evaluation costs over multiple queries.

In our initial implementation of MCDB, random relations are always materialized at query time, because it is the easiest way to ensure that the VG parameterization and other Monte Carlo settings are current, and to avoid potential disk overflow problems. Moreover (given that CPU cycles are very inexpensive these days), we expect it will often be less expensive to parameterize and run the VG function than it will be to read large numbers of materialized tuple bundles from disk. An ideal solution would likely be a hybrid scheme where prematerialization is used for computationally expensive VG functions whose execution would not cause disk overflows, and query-time materialization is used for inexpensive VG functions that produce a lot of data. Developing such a scheme is a topic for future work.

11.3. Parallelizing Instantiate

Since much of MCDB's query processing involves CPU-intensive Monte Carlo computations, there is an opportunity for MCDB to take full advantage of the computational parallelism offered by modern multicore machines. We contrast this with the typical, data-intensive database system, where CPU cycles are invariably wasted due to cross-core contention for shared data access resources such as cache, bus, and RAM. Since MCDB's Monte Carlo processing is encapsulated within the `Instantiate` operation, we spent a fair amount of time designing and implementing a parallel version of `Instantiate` that could efficiently run on many CPU cores. For simplicity, we restrict our discussion to the simple case in which there is exactly one table of input parameters to a single VG function.

We first note that, in our `Instantiate` implementation, the seeded tuples produced by the `FOR EACH` clause must be output from a VG function in the order that they were seeded (see Section D in the online appendix available in the ACM Digital Library). The reason for this requirement is that any "extraneous" seeded-tuple attributes not used by the VG function need to be stripped away. This is done by duplicating the seeded tuple and stripping away the extraneous attributes from one copy. The stripped copy is joined with the VG function's input parameter table in order to pick up any needed parameter values, for example, the `MEAN` and `STD` parameters in the example of Section 4.1, and then is sent to the VG function and instantiated. Execution of the final `SELECT` clause in a `CREATE TABLE` statement might require that some of the extraneous attributes (e.g., `PID` and `GENDER` in the foregoing example) be added back to the instantiated tuple bundle. To do this, we join the instantiated tuple bundle with its unstripped copy. This join uses a sort-merge algorithm with the seed as the join attribute. If the output bundles from the VG function are not sorted according to seed value, then an expensive external sort of the (possibly huge) bundles might be required to execute the merge.

We also note that multiple seeded tuples can share the same seed. For example, if a seeded tuple is joined to a table prior to instantiation (see Section 11.4) and the join is 1-to-m for some $m > 1$, then there will be m joined tuples all sharing the same seed. Thus `Instantiate` will create the same array of realized attribute values for each of the m joined tuples. It follows that a given VG function call can be amortized over multiple tuples, and it can therefore be useful to think in terms of a stream of *seeds*, rather than seeded tuples, flowing into the instantiate operator. A seed can produce multiple instantiated tuple bundles, all of which can be joined with an unstripped copy during the final merge. We sometimes use this "seed" terminology in what follows.

Given c cores, one fairly straightforward parallel implementation of `Instantiate` fires up a new instance of the `Instantiate` code in a worker thread for each new seed encountered, up to c threads in total. Each thread parameterizes its own instance of the VG function, calls the function as needed to produce one or more instantiated tuple bundles, and writes the bundles to a predefined buffer area. Once all c threads complete their work, `Instantiate` sends the instantiated bundles onto the VG function output stream in seed order, where they can be merged with their unstripped copies, and `Instantiate` starts processing the next group of c seeded tuples. The process is then repeated again and again for successive groups of c seeded tuples. The key benefit of this approach is that it trivially allows the tuple bundles to be output from the VG functions in seed order.

Preliminary experiments indicated, however, that this approach performs badly in practice. The key reason for the poor performance is that `Instantiate` cannot send any tuple bundles onto the output stream for merging until all of the threads in the group of c VG function instances complete, so that the slowest VG function instance in the group dictates the group's speed. For example, when rejection algorithms are used to generate random numbers (see query Q4 in Section 13) the execution time for a VG function call can occasionally be longer than the usual time by several orders of magnitude.

We have therefore developed an alternative, more complex scheme that allows for some out-of-order processing of seeds. Instead of blocking until all the worker threads have completed, `Instantiate` blocks whenever it has a new seed ready to be processed by a VG function but none of the existing threads is available. A thread signals `Instantiate` and switches its status to "available" once it has completed all of its Monte Carlo work, that is, all of its VG function invocations. To facilitate output, `Instantiate` maintains a first-in, first-out queue of output buffers. Each output buffer holds all of the tuple bundles corresponding to a single seed value. When a thread finishes, `Instantiate` wakes up, and if a new seed value is ready to be processed, it assigns an output buffer at the tail of the queue to the available thread, which then runs the Monte Carlo iterations required for that seed value. `Instantiate` also looks at the head of the queue to see if the thread associated with the first seed value has completed. If so, it pops the queue by sending the tuple bundles in the associated buffer onto the output stream, and then continues to pop the queue until it finds that the thread associated with the buffer at the head of the queue has not finished its work. The `Instantiate` process then sleeps until the next time that a thread completes.

The use of a queue reduces the idle time of the threads while ensuring that the output bundles are sorted on their seed values, but it is still possible for a very long-running VG function at the head of the queue to cause the queue to grow very long. This situation is handled by not allowing the queue to grow larger than the available main memory. If a slow worker thread at the head of the queue causes problems, `Instantiate` can stop assigning additional work until the worker thread finishes.

To evaluate the performance of this parallelization scheme, we measured the wall-clock running times for 1,000 Monte Carlo iterations of a highly CPU-intensive query as the number of threads increased from 1 to 8. Specifically, we used a query that requires simulation of 1000 Asian call options for 10 customers over 20,000 time steps; see Section 13, query Q8. The results, shown in Figure 4, indicate nearly perfect scale-up.

11.4. Pushing Joins Past Instantiate

A discussion of query compilation and optimization in MCDB is far beyond the scope of this article; even cataloging how the classic algebraic query manipulations change in a

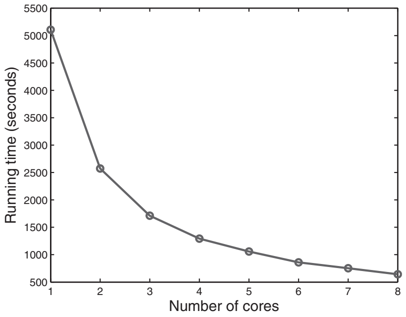

Fig. 4. Performance evaluation of parallel VG function.

system such as MCDB is beyond what we can possibly cover here. The one overriding principle governing query optimization in MCDB, however, is pulling up (i.e., delaying) the `Instantiate` operation as far as possible in a query plan. The reason is simple: because `Instantiate` produces large amounts of data, performing potentially expensive relational operations on the bundles should be avoided by pushing the operations down, past `Instantiate`.

This observation is particularly true in the case of relational join, which may require that the tuples (or tuple bundles) be read and written from disk several times: moving millions or billions of 100KB tuple bundles on and off of disk can be very expensive! Fortunately, there is no reason that a join cannot be executed prior to instantiating the joined seeded tuples. The key requirement is that none of the join attributes be produced as the result of a VG function call within the `Instantiate` operator.

For example, consider the `SBP_DATA` example from Section 4.1. Imagine that we have a deterministic table `TAKES_MED (PID, DRUG, FAMILY)` where a `(PID, DRUG)` combination tells us that the given patient is taking the given drug, and `FAMILY` tells us the family of the drug. Given this table, we want the average blood pressure of people taking some sort of Fluoroquinone, broken down by the specific drug.

```
SELECT AVERAGE(s.SBP)
FROM SBP_DATA s, TAKES_MED t
WHERE t.FAMILY = 'Fluoroquinone'
  AND t.PID = s.PID
GROUP BY t.drug
```

The most efficient query plan in this case is to only run `Instantiate` for those patients who are taking a fluoroquinone. To do this, join `PATIENTS` with `TAKES_MED` on the deterministic `PID` attribute, after filtering `TAKES_MED` so that it only contains fluoroquinones. Then run `Instantiate` on the joined tuples. Thus the join has been pushed down past the `Instantiate`.

`Instantiate` must be implemented with care, however, to allow such optimizations. In particular, recall from Section 11.3 that the stripped copy of a seeded tuple is joined with the VG function input parameter table prior to being sent to the VG function, in order to pick up needed parameter values. Moreover, because of the join, there may be several tuple bundles sharing the same seed. In our example, a single patient might be taking three different fluoroquinones, leading to three tuples containing the seed for the patient's blood pressure VG function. Thus the `MEAN` and `STD` parameters can potentially be picked up three times, wreaking havoc on the VG function call. Thus, if it is possible that the join that is being pushed past `Instantiate` is not a foreign key join, then a duplicate-removal operation must be run on the output of the join between stripped tuples and the VG function input parameter table.

TOT_DI	FRAC
$45M	0.1
$80M	0.3
$140M	0.3
$190M	0.2
$210M	0.1

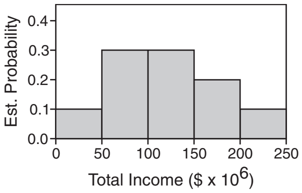

Fig. 5. INCOME table (result of MCDB query), and corresponding approximate histogram.

12. INFERENCE AND ACCURACY

In this section, we discuss in detail how to estimate important features of a query-result distribution and how to provide probabilistic error bounds on such estimates via a pilot-sample approach. For brevity, we focus on a key feature, namely, expected value (or mean). Our general approach, however, applies to a fairly broad range of distribution features; we briefly outline several extensions of our methods. Finally, we indicate potential enhancements to the current functionality.

12.1. MCDB Output

Since MCDB queries are standard SQL queries, a given query result has a well-defined schema R with attributes $R.A_1, R.A_2, \ldots, R.A_k$ for some $k \geq 1$ and, in the absence of uncertainty, the query result is simply a table having this schema. This will be the case for a query that does not involve any random relations or involves only deterministic attributes of random relations. Otherwise, when uncertainty is present and MCDB executes $N > 1$ Monte Carlo iterations, MCDB returns its query results in the form of a table having columns $A_1, A_2, \ldots, A_k$, along with an additional column FRAC that is computed via the MCDB Inference operator (Section 8.4). Each row of the table corresponds to a distinct output tuple produced in the course of the N iterations, with FRAC being the fraction of the N generated query results in which the tuple appears at least once.

Recall, for example, the random relation CUST defined in Section 4.2, and consider the following query, which computes the total disposable income of all customers.

```
QA: SELECT SUM(MONEY) AS TOT_DI FROM CUST
```

If the disposable income of each customer were deterministic (so that no Gamma VG function appeared in the definition of CUST) then the answer to QA would be a one-row, one-column table having the single attribute TOT_DI and containing the (deterministic) total income. Suppose, however, that CUST is indeed defined as in Section 4.2, so that, due to uncertainty, the query answer is no longer a fixed number, but a random variable having a very complicated probability distribution that is unknown to the user.[3] If MCDB executes 1000 Monte Carlo iterations, then the query result might look like the INCOME table in Figure 5. As can be seen, the total income is $45M in 10% of the generated query results, $80M in 30% of the generated query results, and so forth. We can immediately visualize this probability distribution via a histogram as in Figure 5. We can, however, go far beyond graphical displays: the power of MCDB lies in the fact that we can leverage over 65 years of Monte Carlo technology (see Section 2) to make statistical inferences about the distribution of the query answer, about interesting features of this distribution such as means and quantiles, and about the accuracy of

[3] Generally in this setting, NULL values must be treated carefully; see Murthy and Widom [2007] for a discussion of the semantic issues involved.

the inferences themselves. We discuss a couple of particularly important inference problems in the following subsections.

12.2. Estimating the Mean of a Query Result

For aggregation queries such as QA given before, perhaps the most widely studied feature of a query-result distribution is the mean, that is, the expected value, of the query result. Suppose that MCDB executes N Monte Carlo iterations, resulting in observed query-result values $X_1, X_2, \ldots, X_N$. The true mean $\mu \triangleq E[X_i] = \int_{-\infty}^{\infty} x\, dF(x)$ of the query-result distribution F is unknown, but can be estimated by the sample average $\hat{\mu}_N = (1/N)\sum_{i=1}^{N} X_i$. Since $\hat{\mu}_N$ depends on the random output of MCDB, it is itself random, but has several desirable properties. It is *unbiased* in that its expected value equals μ, that is, on average, $\hat{\mu}_N$ gives the correct answer. By the strong law of large numbers, $\hat{\mu}_N$ is *consistent* for μ in that $\hat{\mu}_N$ converges to μ with probability 1 as N increases.

The precision of a random estimator such as $\hat{\mu}_N$ can be measured in several ways. A commonly used and simple measure of precision is the standard deviation, also called the *standard error*, of $\hat{\mu}_N$. The standard error is given by $\sigma/\sqrt{N}$, where $\sigma^2 \triangleq E[(X_i - \mu)^2] = \int_{-\infty}^{\infty} (x-\mu)^2\, dF(x)$ is the true variance of the query-result distribution F. Note that σ^2 can be estimated unbiasedly from the X_i's as $\hat{\sigma}_N^2 = (N-1)^{-1}\sum_{i=1}^{N}(X_i - \hat{\mu}_N)^2$.

We can also obtain probabilistic error bounds. The simplest such bounds are approximate bounds that apply when N is "large", for example, $N \geq 50$ in typical well-behaved problems. These bounds are based on the Central Limit theorem, which asserts that, for large N, the distribution of the estimator $\hat{\mu}_N$ is approximately normal with mean μ and standard deviation $\sigma/\sqrt{N}$ as given earlier. Thus, for a given number N of iterations and a specified probability $p \in (0, 1)$, we can estimate σ by $\hat{\sigma}_N$, draw probabilistic error bars of width $w = z_p \hat{\sigma}_N/\sqrt{N}$ above and below $\hat{\mu}_N$, and then assert that the true mean μ lies within the interval $[\hat{\mu}_N - w, \hat{\mu}_N + w]$ with probability approximately p—here z_p is the $(1 + p)/2$ quantile of the standard normal distribution.

More typically, however, the user has a target maximum error level ϵ, specified a priori. We focus primarily on this scenario, and discuss how to achieve such an error bound by controlling the number of Monte Carlo iterations. A standard calculation shows that, for $p \in (0, 1)$ and $N = z_p^2\sigma^2/(\epsilon\mu)^2$, the estimator $\hat{\mu}_N$ estimates μ to within $\pm 100\epsilon\%$ with probability approximately equal to p. As can been seen, it is impossible to determine N a priori, since μ and σ are unknown. For a given value of p, a standard approach to this problem is estimate these quantities from a "pilot" run. Specifically, we initially run MCDB to create $\tilde{N}$ Monte Carlo replicates, yielding query-result values $\tilde{X}_1, \ldots, \tilde{X}_{\tilde{N}}$. These values can then be used to estimate μ and σ^2 by $\tilde{\mu} = (1/\tilde{N})\sum_{i=1}^{\tilde{N}} \tilde{X}_i$ and $\tilde{\sigma}^2 = (\tilde{N} - 1)^{-1}\sum_{i=1}^{\tilde{N}}(\tilde{X}_i - \tilde{\mu})^2$. Substituting these estimates into our formula for N, we can estimate the required number of replicates in our final, "production" run of MCDB as $N^* = z_p^2\tilde{\sigma}^2/(\epsilon\tilde{\mu})^2$. If we want to control the absolute error, then we simply replace $\epsilon\tilde{\mu}$ by ϵ in the formula for N^*.

To see in more detail how the preceding error bounding technique can easily be implemented in MCDB, recall that, for N Monte Carlo iterations, the output of MCDB for an aggregation query is a table whose ith row is of the form (V_i, ϕ_i), where V_i is the ith distinct aggregate value and ϕ_i is the fraction of the N Monte Carlo iterations for which the query result equals V_i. Some straightforward algebra shows that $\hat{\mu}_N = \sum_{i=1}^{N} V_i\phi_i$ and $\sigma_N^2 = \big(N/(N-1)\big)\sum_{i=1}^{N} V_i^2\phi_i - \hat{\mu}_N^2$. Similar observations hold for $\tilde{\mu}$ and $\tilde{\sigma}^2$. Thus, to estimate the mean of the query-result distribution for the query QA described previously, we set the number of MCDB iterations to $\tilde{N} \geq 50$ and execute QA to obtain an output table INCOME as in Figure 5. We then execute the following ordinary SQL query over the INCOME table.

```
QA2:
WITH STATS(MU, SIG2) AS (
 SELECT SUM(TOT_DI*FRAC), (:N/(:N-1)) *
   SUM(TOT_DI*TOT_DI*FRAC)-SUM(TOT_DI*FRAC)*SUM(TOT_DI*FRAC)
 FROM INCOME)
SELECT MU AS MEAN, SQRT(SIG2) AS STDEV,
  CEILING(:ZP*:ZP*SIG2/(:EPS*:EPS*MU*MU)) AS NUMREP
FROM STATS
```

Here :N, :EPS and :ZP are the number of Monte Carlo iterations used to produce the INCOME table, the desired maximum relative error, and the normal $(1 + p)/2$ quantile corresponding to the desired (approximate) probability guarantee on the error. We then reexecute query QA in MCDB, but this time using NUMREP Monte Carlo iterations. We can then run query QA2 on the resulting INCOME table. This time, MEAN is our final estimate of the mean value of F, which lies within $\pm\epsilon$ of the true value with probability approximately equal to p. For instance, assuming that the INCOME table in Figure 5 represents the result of a pilot run, QA2 would compute $\tilde{\mu} = \$129.5\text{M}$ and $\tilde{\sigma} = \$53.5\text{M}$. If we want to estimate the true mean to within ±10% with 95% probability—so that $\epsilon = 0.10$, $p = 0.95$, and $z_p \approx 1.96$—then the pilot run would indicate that $N^* \approx 263$ replicates are needed in the production run.

Note that if NUMREP $\leq N^*$, then we can use the value of MEAN from the pilot run as our final answer. If Monte Carlo iterations are very expensive, the preceding pilot-run procedure can be modified so that the results of the original $\tilde{N}$ pilot runs are used in computing the final estimate, reducing the number of production runs from N^* to $N^* - \tilde{N}$. This more complex procedure is called "double sampling" [Cox 1952]. These methods can potentially be automated so that the user merely has to specify a desired accuracy and confidence level.

12.3. Extensions and Generalizations

We have focused on a very simple estimation problem to keep the exposition simple and concise. There exists, however, a broad collection of Monte Carlo techniques that a user of MCDB can exploit to perform complex estimation and inference on the query-result distribution. An extensive review of these methods is beyond the scope of this article, so in this section we merely indicate some of the possibilities; see the references given in Section 2 for further discussion.

Estimates and error bounds for other moments of the query-result distribution besides the mean μ, such as the variance, kurtosis, or correlation coefficient, can be obtained in a manner similar to that described in Section 12.2, but with different (often more complicated) estimation formulas. Tuple-inclusion probabilities—which can be viewed as an expected value of an indicator random variable that equals 1 if the tuple is in the query result and equals 0 otherwise—also fall under the purview of these methods, as do quantiles and estimators of the entire query-result distribution function or density function. See Section E of the online appendix available in the ACM Digital Library for several detailed examples.

For more complex features where there exists no analytical formula for the standard error of the estimator, methods such as the bootstrap can be used to estimate the standard error; see, for example, Asmussen and Glynn [2007, Section III.5]. The rough idea is to sample, with replacement, a set of distinct values from the MCDB output table, where the ith row is sampled with probability equal to its FRAC value. An estimate of the feature of interest is then computed from the sampled tuples. This procedure is repeated multiple times, and the standard error of the resulting bootstrapped feature estimates the true standard error.

Besides estimation, we can perform statistical tests of hypotheses such as “the expected value of total income differed from last year to this year.” For example, it is straightforward to write a query that applies a standard paired-sample t-test [Miller 1986, page 1] to an output table with a pair of columns that correspond to the values of these two aggregates over the N Monte Carlo iterations. “Ranking and selection” procedures [Henderson and Nelson 2006, Chapter 17] can potentially be used with MCDB to determine with high probability, for example, the best among alternative business policies.

12.4. Potential Improvements

Many improvements to the currently implemented MCDB inference mechanism are possible. For example, query performance can potentially be enhanced by pushing down the statistical calculations into the query plan. For example, suppose that we know in advance that our sole interest is in computing the estimated mean and variance of the query-result distribution. Then, whenever we instantiate a tuple bundle, we can compute its contribution to these two statistics and then immediately discard the bundle, thereby reducing processing and I/O costs.

Another possible enhancement is to develop an alternative inference operator that is less lossy than the current version. For example, the current inference operator discards information about joint appearance probabilities of tuples, which can be useful for certain queries, for example, calculating the probability that Jane Smith and Jim Thorpe both purchased at least one item in Seattle. One potential approach to this problem (not yet implemented) is to use a modified inference operator that creates an output table in which the `FRAC` column is replaced by a column containing bit vectors of length N, where N, as usual, is the number of Monte Carlo iterations. For the ith row of the table, which contains distinct output tuple t_i and bit vector b_i, we set $b_i[j] = 1$ if and only if t_i appears at least once in the jth generated query-result table. The bit vectors would be formed by eliminating duplicate tuple values in the output, ORing together the *isPresent* vectors of the duplicates. Such output would completely specify joint appearance probabilities. The next level of detail would be to record for each generated query result the multiplicity of each tuple, but it is not clear whether the onerous memory requirements would be justified, or even feasible.

13. EXPERIMENTS

The technical material in this article has focused upon MCDB’s basic Monte Carlo framework, VG function interface, query engine implementation details, and potential ability to handle a broad class of analysis problems on uncertain data. Our experimental study is similarly focused, and has two goals:

(1) To demonstrate examples of nontrivial, “what-if” analyses that are made possible by MCDB.
(2) To determine if this sort of analysis is actually practical from a performance standpoint in a realistic application environment. An obvious upper bound for the amount of time required to compute 100 Monte Carlo query answers is the time required to generate the data and run the underlying database query 100 times. This is too slow. The question addressed is: Can MCDB do much better than this obvious upper bound?

Basic Experimental Setup. We generate a 20GB version of the TPC-H database using TPC-H’s `dbgen` program and use MCDB to run eleven nontrivial “what-if” aggregation queries over this database. Each of the eleven queries is run using one, ten, 100, and

1000 Monte Carlo iterations, and wall-clock running times as well as the query results are collected.

MCDB Software. To process the queries, we use our prototype of the MCDB query-processing engine, which consists of about 20,000 lines of C++ source-code. This multithreaded prototype has full support for the VG function interface described in the article, and contains sort-based implementations of all of the standard relational operations as well as the special MCDB operations. Our MCDB prototype does not yet have a query compiler/optimizer; development of these software components is a goal for future research. The query processing engine's front-end is an MCDB-specific "programming language" that describes the physical query plan to be executed by MCDB.

Hardware Used. We chose our hardware to mirror the dedicated hardware that might be available to an analyst in a small- to medium-sized organization. The eleven queries are run on a dedicated and relatively low-end $3000 server machine with four 160GB ATA hard disks and eight 2.0 GHz cores partitioned over two CPUs. The system has 8GB of RAM and runs Ubuntu Linux.

Queries Tested. The eleven benchmark queries are each computationally expensive, involving joins of large tables, expensive VG-function evaluations, grouping, and aggregation. The SQL for the queries is given in the Appendix.

Query Q1. This query guesses the revenue gain for products supplied by Japanese companies next year (1996), assuming that current sales trends hold. The ratio μ of sales volume in 1995 to 1994 is first computed on a per-customer basis. Then the 1996 sales are generated by replicating each 1995 order a random number of times, according to a Poisson distribution with mean μ. This process approximates a bootstrapping resampling scheme (see Section 12.3). Once 1996 is generated, the additional revenue is computed. In this query, the `Poisson` VG function (which takes as input a single rate parameter) is used.

Query Q2. This query estimates the number of days until all orders that were placed today are delivered. Using past data, the query computes the mean and variance of both time-to-shipment and time-to-delivery for each part. For each order placed today, instances of these two random delays are generated according to discretized gamma distributions with the computed means and variances. Once all of the times are computed, the maximum duration is selected. In this query, the `DiscGamma` VG function (for "discretized gamma") is used. This function takes two parameters, the mean and variance of the desired gamma distribution.

Query Q3. One shortcoming of the TPC-H schema is that, for a given supplier and part, only the current price is maintained in the database. Thus, it is difficult to ask, "What would the total amount paid to suppliers in 1995 have been if we had always gone with the most inexpensive supplier?" Query Q3 starts with the current price for each item from each supplier and then performs a random walk to guess prices from December, 1995 back to January, 1995. The relative price change per month is assumed to be normally distributed with a mean of -0.02 and a variance of 0.04. The most inexpensive price available for each part is then used. In this query, the `RandomWalk` VG function is used. This takes as input a starting point for the walk (a starting price in this case), the starting month for the walk, the number of months to walk backwards in time, and the mean and variance of the change in price each month. It then walks backwards in time, emitting a possible price for each month.

Query Q4. This is the query mentioned in Section 1, which estimates the effect of a 5% customer price increase on an organization's profits. The Bayesian posterior distribution function used to predict a customer's demand at a new price does not have a closed-form representation, and so Monte Carlo methods *must* be used.

At a high level, this VG function works as follows. For a given part that can be purchased, denote by D_p a given customer's random demand for this part when the price equals p. A prior distribution for D_p is used that is the same for all customers. Bayesian methods are used to obtain a posterior, customer-specific distribution for D_p (for all values of p) by combining the generic prior distribution with our knowledge of the actual price p^* offered to the customer, and the customer's demand d^*. The inner workings of the VG function are described in more detail in the Appendix, and are encapsulated within the Bayesian VG function.

Query Q5. This query and the following query are designed to demonstrate that MCDB can easily implement the tuple-appearance uncertainty model of Dalvi and Suciu [2007b], and to investigate how well MCDB performs when it is used for such a model. Both queries are closely based upon test queries that were run by Dalvi and Suciu in their paper. Both of them make use of the Bernoulli VG function, which takes as input a probability p and outputs either a true or a false, with $Pr[\texttt{true}] = 1 - Pr[\texttt{false}] = p$.

Q5 is based upon query 3 from the TPC-H benchmark. This query estimates the revenue for orders from the HOUSEHOLD market segment that were made in March 1995 and shipped by the end of the month. The VG function in Q5 implements a "fuzzy" equality operator that compares the market segment string with the literal HOUSEHOLD. If the market segment is exactly the string HOUSEHOLD, then the probability of a match is one. As the edit distance between the market segment and HOUSEHOLD increases, the probability of a match decreases. This is implemented by computing the edit distance d between the market segment and the literal, and setting the probability of a match to be $p = 2 - 2\Phi(d)$, where Φ is the standard normal cumulative density function. For example, an edit distance equal to 1 produces a match probability of 0.318, and a distance equal to 2 produces a probability of 0.0456. The probability of the order being made and shipped as specified is computed in a similar fashion, as detailed in the Appendix.

Query Q6. This is similar to query 6 from the TPC-H benchmark. This query estimates the total revenue for all the 1996 shipments with discounts between 3% and 5% and a quantity of around 25 units, again using the Dalvi-Suciu-style model. In this query, the ship date, discount, and quantity are all compared to literal target values using a fuzzy match, using a methodology similar to the one from Q5.

Query Q7. The motivation behind this query is to be able predict the change in revenue if shipments are processed more quickly than they were processed in reality, since it is reasonable to expect that customers are prone to rejecting a shipment if it takes too long to arrive.

First, a logistic regression model is learned (offline) using the returnflag attribute of lineitem. This model relates the probability that an item is returned to the length of time that is required to process the shipment. This model is embodied in the user-defined function logistic, which returns the probability that the item will be returned, given the shipping time. These probabilities, along with the Bernoulli VG function, are used to predict the change in revenue if the shipping times are cut by 25%.

Query Q8. This query simulates the possible profits at some future time T for a set of customers, each of whom holds some number of call options. The profits associated with a given option are predicted using a stochastic model. This query is unlike any of the others in the sense that the stochastic stock-price simulation performed by the VG function (rather than any of the required database processing) is by far the most expensive component of the query.

In the query, there are 10,000 customers, and each customer owns between 1 and 100 call options comprising one to five distinct option types. There are 1000 possible call-option types. All customers in the database have bought the options at the same time. The owner of an option of type i has the right to purchase a share of an underlying stock at a given *strike price* K_i at a specified time T (assumed the same for all options considered). The owner of the stock can collect a payoff at time T, denoted $P_i(T)$, which depends on the value of the stock over the interval $[0, T]$. Thus, if $P_i(T) > K_i$, the customer can purchase the stock and immediately collect the payoff, for a profit of $P_i(T) - K_i$; if $P_i \leq K_i$, then the option is worthless, and the customer makes 0 profit. Thus the value of the option type i at time T is $max\big(P_i(T) - K_i, 0\big)$. The total profit for a customer who owns n_i options of type i ($i = 1, 2, \ldots, m$) is $\sum_{i=1}^{m} n_i \times \max\big(P_i(T) - K_i, 0\big)$.

If we knew $P_i(T)$, the profit for each customer would be easy to compute. However, in this query we do not know $P_i(T)$. For our example, we assume that all option types are Asian call options, and we use a modified Black-Scholes stochastic model of stock-price evolution to simulate values of $P_i(T)$. We simulate the successive prices of each stock at 20,000 discrete time points in $[0, T]$. This simulation is encapsulated in the `Val_Comp` VG function; see the Appendix for details.

Query Q9. This query estimates the potential profit of a marketing campaign. The query is particularly difficult to evaluate because (unlike any of the other queries) it requires a nested-loops join involving a random table. This is necessary since the query requires a spatial join. The query stochastically creates customers (who have random locations) and then joins those customers with the table determining how much money is spent on advertising in each region. The join predicate makes use of the physical location of each customer.

Specifically, suppose that for each one of a company's existing customers in a specified region, marketing research indicates that there are approximately 1.5 similar customers nearby who could be reached via a targeted marketing campaign. Also suppose that the (deterministic) set of existing customers is specified by the TPC-H `customer` table, augmented with two location attributes $X, Y \in [0, 10]$.[4] The company has designed a specific campaign that divides the region into subregions and allocates a specified amount of advertising money to each region. Each potential customer has a random threshold amount of advertising money that must be spent in his/her subregion for him/her to become an actual customer.

We define a random table `potential_customer` using a VG function `CustGen` that generates potential customers as follows. First, each existing customer c_i is replicated N_i times, where $N_i \sim$ Poisson; the location difference between each replicate of c_i and c_i itself is generated independently from a bivariate normal distribution with mean $[0.25, 0.25]^T$, variance $[1, 1]^T$, and correlation coefficient 0.3. Next, the threshold value for each potential customer is independently generated from a gamma distribution with shape = scale = 3. The query of interest joins `potential_customer` with a table called `marketcampaign` that specifies the amount of money to be spent in each subregion. Those potential customers that are exposed to enough advertising are then used to compute the potential profit.

In this query, the `CustGen` VG function has three parameters: the rate parameter given to the Poisson distribution that tells the VG function how many times the customer should be replicated, as well as the physical location of the customer.

Query Q10. In this query, we estimate the revenue obtained for all the customers from Japan, under the assumption that the customer listed on each tuple in the `orders` table may be wrong, possibly due to an error in the data integration process. To quantify this error, a new relation `error_custkey(old_custkey, custkey, prob)` is created such that, for each customer key `old_custkey` from the original `customer` relation, we have

[4] We generate the values of each pair (X, Y) from a bivariate normal distribution with mean $[5, 5]^T$, variance $[1, 1]^T$, and correlation coefficient 0.4. Any generated values that are smaller than 0 or larger than 10 are replaced by 0 and 10, respectively.

	Time required				Iterations required		
Query	1 iter	10 iters	100 iters	1000 iters	10% err	1% err	0.1% err
Q1	26 min	27 min	27 min	28 min	50	50	50 + 13
Q2	40 min	41 min	40 min	41 min	50	50 + 292	50 + 34,240
Q3	42 min	42 min	101 min	318 min	50	50	50
Q4	40 min	39 min	52 min	213 min	50	50	50
Q5	18 min	18 min	19 min	25 min	50	50 + 1,842	50 + 189,160
Q6	34 min	35 min	36 min	85 min	50	50	50
Q7	30 min	31 min	31 min	45 min	50	50 + 350	50 + 39,909
Q8	1 min	1 min	7 min	71 min	50	50	50 + 1,296
Q9	30 min	53 min	88 min	835 min	50	50	50 + 404
Q10	18 min	18 min	18 min	18 min	50	50	50 + 429
Q11	18 min	18 min	18 min	17 min	50	50	50

Fig. 6. Benchmarking results. The left-hand side of the table shows the time required to execute 1, 10, 100, and 1000 Monte Carlo iterations, for each of the eleven queries. The right-hand side shows how many Monte Carlo iterations would be required to estimate the mean (expected value) of the query result at 10% error, 1% error, and 0.1% error, with 95% probability. This is computed using a pilot-sampling approach, as described in Section 11.2. The pilot-sampling approach first runs 50 Monte Carlo iterations (the "pilot sample") and uses this to compute the number of iterations required to reach the desired accuracy. If any additional iterations are required, they are taken as a separate step. Hence, "50 + 292" means that in additional to the pilot sample, a second step with 292 additional Monte Carlo trials is required.

a potentially "correct" customer key custkey and a probability prob of custkey being, in reality, the correct one.

The error_custkey parameter table is created externally by sorting the customer keys lexicographically, and then setting the probability that observed key k should have been recorded as $k + i$ to be $\frac{1}{2^{k+i}}$ (i is only allowed to range from 0 to 10; the remaining small unattached probability is allocated to all keys proportionally).

This query uses the DiscreteChoice VG function, which takes as input a set of items (and associated weights) from which it will make a random selection. Each item in this case is a possible customer key.

Query Q11. This query is closely related to Q7. However, instead of using a logistic regression model, a resampling scheme is used to estimate the revenue if the shipment time of each 1995 order is cut in half. Specifically, we "guess" whether or not a faster shipment would be rejected by the customer as follows. The DiscreteChoice VG function is fed, for each 1995 lineitem, the set of all shipments for the same part that took exactly half of the observed shipping time. It then selects one of those shipments at random and uses the returnflag attribute to determine whether or not the faster shipment is rejected. Thus the probability of rejection is precisely the fraction of speedy deliveries that were actually rejected.

Results. The results obtained by running the eleven queries are given in Figure 6. To put the running times in perspective, we ran a foreign key join over partsupp, lineitem, and orders in Postgres (out of the box, with no tuning), and killed the query after waiting more than 2.5 hours for it to complete. A commercial system would probably be much faster, but this shows that MCDB times are not out of line with what one may expect from a classical query processing engine.

In terms of the running times required, the results are very encouraging. For queries Q1, Q2, Q5, Q7, Q10, and Q11, there is only a small-to-negligible dependence of running time on the number of Monte Carlo iterations run. What this means is that for those queries, all of the other fixed costs associated with running the query dominate: the time to parameterize the VG functions, the fixed-cost latency associated with loading the tuples/tuples bundles from disk (i.e., seek time) or from memory and onto the CPU (i.e., cache miss stalls), the cost of running selections and joins on constant attributes, and so forth. The variable costs that increase with additional Monte Carlo iterations seem to be relatively unimportant; these include both the cost of running the Monte Carlo simulations and the extra transfer times associated with moving large tuple bundles around. As a result, MCDB is much faster than naively regenerating the data and rerunning the underlying query many times. Of course, if the number of Monte Carlo iterations were increased so as to be very, very large, the running time would increase linearly with the increasing iterations.

It is instructive to consider in detail those queries for which additional Monte Carlo trials *do* significantly affect query running time. In Q3, the VG function produces a huge amount of data (a random walk consisting of twelve tuple bundles for each tuple in `partsupp`), and the tuple bundles containing this data are fed into a subsequent, disk-based `GROUP BY` operation. This `GROUP BY` incurs hundreds of gigabytes of I/O, with the amount of I/O growing linearly with each additional Monte Carlo trial. Hence there is a strong dependence of running time on the number of Monte Carlo trials.

In Q4, the VG function itself is quite expensive, and the cost to run the simulation itself tends to dominate. Since this cost increases linearly with the number of trials, the overall query cost shows moderate dependence on the number of trials run. This is also the case in Q8 (the option price query), where the data set itself is quite small but the VG function encapsulates a very complex computation.

Q9 is the one example where MCDB's query processing engine fails to do radically better than simply regenerating the database and rerunning the underlying query multiple times. The reason is that Q9 requires a nested-loops join of the stochastic table `potential_customer` with the `marketcampaign` table. The boolean condition associated with this nested-loops join references several stochastic attributes from `potential_customer`. As a result, every Monte Carlo value in `potential_customer` is referenced many times in every Monte Carlo iteration. Thus, the extra cost associated with each additional Monte Carlo trial is very high, and there is a very strong dependence upon the number of Monte Carlo trials. In fact, the running time associated with 1,000 trials is approximately 9.5 times as large as the time associated with 100 trials.

Still, we view Q9 as the "exception that proves the rule." Overall, MCDB is able to very successfully ensure that many Monte Carlo trials can be run with little more expense than simply running one or two trials. (Even for Q9, the time required for MCDB to run 1,000 trials is still much less than the time required to run this query 1,000 times with one trial per run.)

We also make a few remarks regarding the number of Monte Carlo trials required to achieve high accuracy. For each of the eleven queries, the pilot sample alone suffices to attain an error of ±10% when estimating the query-result distribution's mean. For an error of ±1%, additional samples are required in only three of eleven queries, and even in these three cases, the number of iterations required is still reasonable.

These results do not guarantee that MCDB can always reach ±1% error in a reasonable amount of time; a particular model and dataset may have much higher variance than those that we tested here. Still, we are encouraged because we view ±1% error as a reasonable goal. After all, MCDB is only "accurate" to the extent that the underlying model is accurate, and the modeling process itself is likely to introduce quite significant biases and errors due to the choice of model and fitting of model parameters. It is of questionable utility to obtain ±0.1% error (which sometimes *does* require an unreasonable number of samples; nearly 200,000 in Q5) when the modeling process itself is seldom that accurate.

14. CONCLUSIONS

This article describes an initial attempt to design and prototype a Monte-Carlo-based system for bringing sophisticated stochastic analytics close to the data. The MCDB approach (which uses the standard relational data model, VG functions, and parameter tables) provides a powerful and flexible framework for representing uncertainty. Our experiments indicate that our new query-processing techniques permit handling of uncertainty at acceptable overheads relative to traditional systems.

Much work remains to be done, and there are many possible research directions. Some important issues we are exploring include a comprehensive treatment of query compilation and optimization, exploitation of computing clusters to parallelize Monte Carlo computations, and the use of advanced Monte Carlo methods to facilitate risk assessment and handling of recursive stochastic models. Some progress has recently been made along these lines [Arumugam et al. 2010; Perez et al. 2010; Xu et al. 2009]. We also hope to extend the techniques and ideas developed here to other types of data, such as uncertain XML [Kimelfeld et al. 2009]. Overall, by bringing stochastic analytics into the database, MCDB has the potential to facilitate real-world risk assessment and decision making under uncertainty, both key tasks in a modern enterprise.

APPENDIX: EXPERIMENTAL QUERIES

Query Q1.

```
CREATE VIEW from_japan AS
SELECT *
FROM nation, supplier, lineitem, partsupp
WHERE n_name='JAPAN' AND s_suppkey=ps_suppkey AND ps_partkey=l_partkey
  AND ps_suppkey=l_suppkey AND n_nationkey=s_nationkey

CREATE VIEW increase_per_cust AS
SELECT o_custkey AS custkey, SUM(yr(o_orderdate)-1994.0)/
  SUM(1995.0-yr(o_orderdate)) AS incr
FROM ORDERS
WHERE yr(o_orderdate)=1994 OR yr(o_orderdate)=1995
GROUP BY o_custkey

CREATE TABLE order_increase AS
  FOR EACH o in ORDERS
    WITH temptable AS Poisson(
      SELECT incr
      FROM increase_per_cust
      WHERE o_custkey=custkey AND yr(o_orderdate)=1995)
    SELECT t.value AS new_cnt, o_orderkey
    FROM temptable t

SELECT SUM(newRev-oldRev)
FROM (SELECT l_extendedprice*(1.0-l_discount)*new_cnt AS newRev,
        (l_extendedprice*(1.0-l_discount)) AS oldRev
      FROM order_increase, from_japan
      WHERE l_orderkey=o_orderkey)
```

Query Q2.

```
CREATE VIEW orders_today AS
SELECT *
FROM orders, lineitem
WHERE o_orderdate=today AND o_orderkey=l_orderkey

CREATE VIEW params AS
SELECT AVG(l_shipdate-o_orderdate) AS ship_mu,
  AVG(l_receiptdate-l_shipdate) AS arrv_mu,
  STD_DEV(l_shipdate-o_orderdate) AS ship_sigma,
  STD_DEV(l_receiptdate-l_shipdate) AS arrv_sigma,
  l_partkey AS p_partkey
FROM orders, lineitem
WHERE o_orderkey=l_orderkey
GROUP BY l_partkey

CREATE TABLE ship_durations AS
  FOR EACH o in orders_today
    WITH gamma_ship AS DiscGamma(
      SELECT ship_mu, ship_sigma
      FROM params
      WHERE p_partkey=l_partkey)
    WITH gamma_arrv AS DiscGamma(
      SELECT arrv_mu, arrv_sigma
      FROM params
      WHERE p_partkey=l_partkey)
  SELECT gs.value AS ship, ga.value AS arrv
  FROM gamma_ship gs, gamma_arrv ga

SELECT MAX(ship+arrv)
FROM ship_durations
```

Query Q3.

```
CREATE TABLE prc_hist(ph_month, ph_year, ph_prc, ph_partkey) AS
  FOR EACH ps in partsupp
    WITH time_series AS
      RandomWalk(VALUES (ps_supplycost,12,"Dec",1995,-0.02,0.04))
    SELECT month, year, value, ps_partkey
    FROM time_series ts

CREATE VIEW best_price AS
SELECT MIN(ph_prc) AS min_prc, ph_month, ph_year, ph_partkey
FROM prc_hist
GROUP BY ph_month, ph_year, ph_partkey

SELECT SUM(min_prc*l_quantity)
FROM best_price, lineitem, orders
WHERE ph_month=month(o_orderdate) AND l_orderkey=o_orderkey
  AND yr(o_orderdate)=1995 AND ph_partkey=l_partkey
```

Query Q4.

```
CREATE VIEW params AS
SELECT 2.0 AS p0shape, 1.333*AVG(l_extendedprice*(1.0-l_discount))
  AS p0scale, 2.0 AS d0shape, 4.0*AVG(l_quantity) AS d0scale,
  l_partkey AS p_partkey
FROM lineitem l
GROUP BY l_partkey

CREATE TABLE demands (new_dmnd, old_dmnd, old_prc, new_prc,
    nd_partkey, nd_suppkey) AS
  FOR EACH l IN (SELECT * FROM lineitem, orders
                 WHERE l_orderkey=o_orderkey AND
                 yr(o_orderdate)=1995)
    WITH new_dmnd AS Bayesian (
      (SELECT p0shape, p0scale, d0shape, d0scale
       FROM params
       WHERE l_partkey = p_partkey)
      (VALUES (l_quantity, l_extendedprice*(1.0-
       l_discount))/l_quantity, l_extendedprice*
       1.05*(1.0-l_discount)/l_quantity))
    SELECT nd.value, l_quantity, l_extendedprice*
      (1.0-l_discount))/ l_quantity, 1.05*
      l_extendedprice*(1.0-l_discount)/l_quantity,
      l_partkey, l_suppkey
    FROM new_dmnd nd

SELECT SUM (new_prf-old_prf)
FROM (
  SELECT
    new_dmnd*(new_prc-ps_supplycost) AS new_prf
    old_dmnd*(old_prc-ps_supplycost) AS old_prf
  FROM partsupp, demands
  WHERE ps_partkey=nd_partkey AND
    ps_suppkey=nd_suppkey)
```

Query Q5.

```
CREATE TABLE fuzzy_lineitem AS
  FOR EACH l in lineitem
    WITH res AS Bernoulli(
      VALUES(NormCDF(0,10,'March 31, 1995'-l_shipdate)))
  SELECT l_extendedprice, l_discount, l_orderkey,
    value AS l_ispres
  FROM res

CREATE TABLE fuzzy_cust AS
  FOR EACH c in customer
    WITH res AS Bernoulli(
      VALUES(2.0-2.0*NormCDF(0,1,ed('HOUSEHOLD',mktsegment))))
  SELECT c_custkey, value AS c_ispres
  FROM res

CREATE TABLE fuzzy_order AS
  FOR EACH o in orders
    WITH res AS Bernoulli(
      VALUES(NormCDF(0,10,o_orderdate-'March 1, 1995')))
  SELECT o_orderkey, value AS o_ispres
  FROM res

SELECT l_orderkey, SUM(l_extendedprice*(1-l_discount)) as revenue
FROM fuzzy_lineitem, fuzzy_customer, fuzzy_orders
WHERE c_custkey = o_orderkey AND o_orderkey = l_orderkey AND
  o_ispres = true AND c_ispres = true AND l_ispres = true
GROUP BY l_orderkey
```

Query Q6.

```
CREATE TABLE fuzzy_lineitem AS
  FOR EACH l IN lineitem
    WITH res1 AS Bernoulli(
      VALUES(2.0-2.0*NormCDF(0, 180, abs(l_shipdate-'Jun 1, 1996'))))
    WITH res2 AS Bernoulli(
      VALUES(2.0-2.0*NormCDF(0.0, 0.01, abs(l_discount-0.04))))
    WITH res3 AS Bernoulli(
      VALUES(2.0-2.0*NormCDF(0.0, 10, abs(l_quantity-25))))
  SELECT l_extendedprice, l_discount
  FROM res1, res2, res3
  WHERE res1.value = true AND res2.value = true AND res3.value = true

SELECT SUM((l_extendedprice * l_discount) * newCount) AS revenue
FROM fuzzy_lineitem;
```

Query Q7.

```
CREATE VIEW order_times AS
SELECT (l_receiptdate - o_orderdate) AS days,
  (l_receiptdate - o_orderdate)*0.75 AS less_days,
FROM lineitem, orders
WHERE l_orderkey = o_orderkey

CREATE TABLE shipped AS
  FOR EACH ot IN order_times
    WITH res1 AS Bernoulli(VALUES(Logistic(days)))
    WITH res2 AS Bernoulli(VALUES(Logistic(less_days)))
  SELECT res1.value AS ret_orig, res2/value AS ret_new
  FROM res1, res2
SELECT SUM((l_extendedprice*l_discount)*ret_orig) -
       SUM((l_extendedprice*l_discount)*ret_new) AS rev_diff
FROM shipped
```

Query Q8.

```
CREATE TABLE option_val(oid,val) AS
 FOR EACH o IN option
  WITH oval AS ValComp(VALUES(o.initval, o.r, o.sigma, o.k, o.m, o.t))
  SELECT o.oid, v.value
  FROM oval v

SELECT c.cid, SUM(c.num * ov.val)
FROM cust c, option_val ov
WHERE c.oid = ov.oid
GROUP BY c.cid
```

Query Q9.

```
CREATE TABLE potential_customer(custkey, x, y, adsig) AS
 FOR EACH c in customer
  WITH custgen AS CustGen (VALUES(1.5, c.x, c.y))
  SELECT c_custkey, cg.x, cg.y, cg.adsig
  FROM custgen cg

SELECT SUM(l_extendedprice*(1.0-l_discount))
FROM lineitem c, orders o, marketcampaign mc, potential_customer pc
WHERE mc.xmin < pc.x AND mc.xmax > pc.x AND mc.ymin < pc.y
      AND mc.ymax > pc.y AND mc.adsig > pc.adsig
      AND pc.custkey = o.o_custkey AND o.o_orderkey = l.l_orderkey
```

Query Q10.

```
CREATE VIEW from_japan AS
SELECT c_custkey
FROM customer, nation
WHERE n_nationkey = c_nationkey AND n_name = 'JAPAN'

CREATE TABLE fixed_cust AS
 FOR EACH o IN orders
  WITH newcustkey AS DiscreteChoice(
    SELECT custkey, probability
    FROM error_custkey
    WHERE old_custkey = o_custkey)
  SELECT value AS o_newcustkey, o_orderkey
  FROM newcustkey

SELECT SUM(l_extendedprice * (1.0 - l_discount))
FROM lineitem, fixed_cust, from_japan
WHERE l_orderkey = o_orderkey AND o_newcustkey = c_custkey
```

Query Q11.

```
CREATE VIEW inner_lineitem AS
SELECT l_partkey AS i_partkey, ((l_receiptdate - l_shipdate) / 2) AS i_duration;

CREATE TABLE likely_incomes AS
 FOR EACH l IN lineitem
   WITH newres AS DiscreteChoice(
     SELECT l_returnflag
     FROM inner_lineitem
     WHERE i_duration = (l.receiptdate - l_shipdate) AND
            i_partkey = l.l_partkey)
 SELECT (l_extendedprice * (1.0 - l_discount)) AS income
 FROM newres
 WHERE newres.value = 'N'

SELECT SUM(income) AS revenue
FROM likely_incomes;
```

Details of VG function for query Q4. As discussed in Section 1, the prior distributions of P_0 and D_0 are Gamma(k_p, θ_p) and Gamma(k_d, θ_d), respectively, with P_0 and D_0 mutually independent. We choose shape parameters $k_p = k_d = 2.0$ and scale parameters $\theta_p = \frac{4}{3}\times$ (the average price), and $\theta_d = 4\times$ (the average demand), where the average price and demand are computed over all of the existing records. Given our choice of k_p and k_d, our subsequent choice of θ_p and θ_d ensures that the average price and demand over all customers for a given item actually falls on the most likely demand curve; this most-likely curve is depicted in Figure 7.

Given the observation (p^*, d^*) for a customer, we generate (P_0, D_0) from the posterior distribution described in Section 1 and then compute the demand at the new price $p = 1.05p^*$ as $D_p = (D_0/P_0)(P_0 - p)$. To this end, we can use Bayes' rule to write down an expression for the posterior density, up to a normalization factor. Although we cannot compute the normalizing constant—and hence the demand-function density—in closed form, we can generate a pair (P_0, D_0) according to this density, using a "rejection sampling" algorithm. The VG function for customer demand, then, determines demand for the 5% price increase essentially by: (1) using Bayes' rule to determine the parameters of the rejection sampling algorithm, (2) executing the sam-

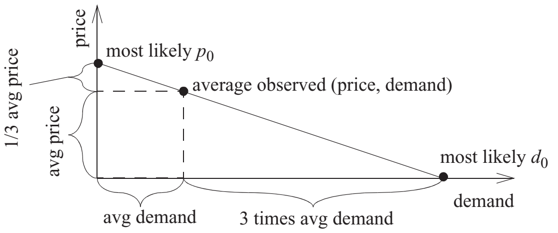

Fig. 7. Most likely demand curve under prior distribution.

pling algorithm to generate (P_0, D_0), and then (3) computing the predicted demand D_p.

In more detail, let $g(x; k, \theta) = x^{k-1}e^{-x/\theta}/\theta^k\Gamma(k)$ be the standard gamma density function with shape parameter k and scale parameter θ, and set $g_p(x) = g(x; k_p, \theta_p)$ and $g_d(x) = g(x; k_d, \theta_d)$. Then the prior density function for (P_0, D_0) is $f_{P_0,D_0}(x, y) = g_p(x)g_d(y)$. If a demand curve passes through the point (p^*, d^*), then P_0 and D_0 must be related as follows: $P_0 = p^*D_0/(D_0 - d^*)$. Let $h(x, y) = 1$ if $x \geq d^*$ and $y = p^*x/(x - d^*)$; otherwise, $h(x, y) = 0$. For $x \geq d^*$, Bayes' theorem implies that

$$\begin{aligned} &P\left\{D_0 = x, P_0 = y \mid P_0 = p^*D_0/(D_0 - d^*)\right\} \\ &\quad \propto P\left\{P_0 = p^*D_0/(D_0 - d^*) \mid D_0 = x, P_0 = y\right\} \times P\left\{D_0 = x, P_0 = y\right\} \\ &\quad = h(x, y)g_d(x)g_p(y) = h(x, y)g_d(x)g_p\big(p^*x/(x - d^*)\big). \end{aligned}$$

In other words, $h_d(x) = cg_d(x)g_p\big(p^*x/(x - d^*)\big)$ is the posterior density of D_0—where c is a constant such that $\int_{x=d^*}^{\infty} h_d(x)\,dx = 1$—and P_0 is completely determined by D_0. The normalization constant c has no closed-form representation. Our VG function generates samples from h_d using a simple, approximate rejection algorithm that avoids the need to compute c. Specifically, we determine a value $x_{\max}$ such that $\int_{x=d^*}^{x_{\max}} h_d(x)\,dx \approx 1$, and also numerically determine the point x^* at which $c^{-1}h_d$ obtains its maximum value. The rejection algorithm generates two uniform random numbers U_1 and U_2 on $[0, 1]$, sets $X = d^* + U_1(x_{\max} - d^*)$, and "accepts" X if and only if $c^{-1}h_d(x^*)U_2 \leq c^{-1}h_d(X)$; if the latter inequality does not hold, then X is "rejected." This process is repeated until a value of X is accepted, and this accepted value is returned as the sample from h_d. The correctness of the rejection algorithm follows by a standard argument [Devroye 1986]. Once we have generated a sample D_0 from h_d, we determine P_0 deterministically as $P_0 = p^*D_0/(D_0 - d^*)$. Finally, $D_p = (D_0/P_0)(P_0 - p)$.

Asian Call Options with Black-Scholes Dynamics for query Q8. Let $S_i(t)$ denote the value of the underlying stock at time t; we assume that the stock evolves according to the Black-Scholes stochastic differential equation: $dS_i(t)/S_i(t) = r_i\,dt + \sigma_i\,dW_i(t)$, where $W_1, W_2, \ldots$ are independent standard Brownian motions, r_i is the risk-free interest rate, and σ_i is the "volatility." The payoff for an Asian option is the average price over a sequence of $m \geq 1$ measurement times in $[0, T]$, assumed, for simplicity, to be equally spaced. Thus $P_i(T) = m^{-1}\sum_{j=1}^{m} S_i(t_j)$, where $t_j = (j/m)T$ for $j = 1, 2, \ldots, m$. For convenience, set $t_0 = 0$. The initial stock price $S_i(t_0) = S_i(0)$ is assumed known. Under the assumed stock dynamics, we have $S_i(t_j) = S_i(t_{j-1})\exp\big((r_i - 0.5\sigma_i^2)(t_j - t_{j-1}) + \sigma_i\sqrt{(t_j - t_{j-1})}Z_{ij}\big)$ for $j = 1, 2, \ldots, m$, where $Z_{i1}, Z_{i2}, \ldots, Z_{im}$ is a sequence of independent and identically distributed normal random variables with mean 0 and variance 1.

REFERENCES

Agrawal, P., Benjelloun, O., Sarma, A. D., Hayworth, C., Nabar, S. U., Sugihara, T., and Widom, J. 2006. Trio: A system for data, uncertainty, and lineage. In *Proceedings of the International Conference on Very Large Databases (VLDB'06).*

Alur, N. R., Haas, P. J., Momiroska, D., Read, P., Summers, N. H., Totanes, V., and Zuzarte, C. 2002. *DB2 UDB's High Function Business Intelligence in e-Business*. IBM Redbook Series.

Andritsos, P., Fuxman, A., and Miller, R. J. 2006. Clean answers over dirty databases: A probabilistic approach. In *Proceedings of the International Conference on Data Engineering (ICDE'06).* 30.

Antova, L., Jansen, T., Koch, C., and Olteanu, D. 2008. Fast and simple relational processing of uncertain data. In *Proceedings of the International Conference on Data Engineering (ICDE'08).* 983–992.

Antova, L., Koch, C., and Olteanu, D. 2007. MayBMS: Managing incomplete information with probabilistic world-set decompositions. In *Proceedings of the International Conference on Data Engineering (ICDE'07).* 1479–1480.

ApacheMahout. 2010. Apache mahout. http://lucene.apache.org/mahout/

Arumugam, S., Jampani, R., Perez, L. L., Xu, F., Jermaine, C. M., and Haas, P. J. 2010. MCDB-R: Risk analysis in the database. In *Proceedings of the International Conference on Very Large Databases (VLDB'10).* 782–793.

Asmussen, S. and Glynn, P. W. 2007. *Stochastic Simulation: Algorithms and Analysis*. Springer.

Barbara, D., Garcia-Molina, H., and Porter, D. 1992. The management of probabilistic data. *IEEE Trans. Knowl. Data Engin. 4*, 5, 487–502.

Benjelloun, O., Sarma, A. D., Halevy, A. Y., Theobald, M., and Widom, J. 2008. Databases with uncertainty and lineage. *VLDB J. 17*, 2, 243–264.

Biller, B. and Nelson, B. L. 2003. Modeling and generating multivariate time-series input processes using a vector autoregressive technique. *ACM Trans. Model. Comput. Simul. 13*, 3, 211–237.

Blei, D. M., Griffiths, T. L., Jordan, M. I., and Tenenbaum, J. B. 2003. Hierarchical topic models and the nested chinese restaurant process. In *Proceedings of the Conference on Advances in Neural Information Processing Systems (NIPS'03).*

Cheng, R., Singh, S., and Prabhakar, S. 2005. U-DBMS: A database system for managing constantly-evolving data. In *Proceedings of the International Conference on Very Large Databases (VLDB'05).* 1271–1274.

Chu, C.-T., Kim, S. K., Lin, Y.-A., Yu, Y., Bradski, G. R., Ng, A. Y., and Olukotun, K. 2006a. Map-Reduce for machine learning on multicore. In *Proceedings of the Conference on Advances in Neural Information Processing Systems (NIPS'06).* 281–288.

Chu, D., Deshpande, A., Hellerstein, J. M., and Hong, W. 2006b. Approximate data collection in sensor networks using probabilistic models. In *Proceedings of the International Conference on Data Engineering (ICDE'06).* 48.

Cohen, J., Dolan, B., Dunlap, M., Hellerstein, J. M., and Welton, C. 2009. MAD skills: New analysis practices for big data. *Proc. VLDB 2*, 2, 1481–1492.

Cox, D. R. 1952. Estimation by double sampling. *Biometrika 39*, 3-4, 217–227.

Dalvi, N. N., Re, C., and Suciu, D. 2009. Probabilistic databases: Diamonds in the dirt. *Comm. ACM 52*, 7, 86–94.

Dalvi, N. N. and Suciu, D. 2007a. The dichotomy of conjunctive queries on probabilistic structures. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Princeiples of Database Systems (PODS'07).* 293–302.

Dalvi, N. N. and Suciu, D. 2007b. Efficient query evaluation on probabilistic databases. *VLDB J. 16,* 4, 523–544.

Dalvi, N. N. and Suciu, D. 2007c. Management of probabilistic data: Foundations and challenges. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'07).* 1–12.

Das Sarma, A., Benjelloun, O., Halevy, A. Y., Nabar, S. U., and Widom, J. 2009. Representing uncertain data: Models, properties, and algorithms. *VLDB J. 18*, 5, 989–1019.

Das Sarma, A., Theobald, M., and Widom, J. 2008. Exploiting lineage for confidence computation in uncertain and probabilistic databases. In *Proceedings of the International Conference on Data Engineering (ICDE'08).* 1023–1032.

Deshpande, A. and Madden, S. 2006. MauveDB: Supporting model-based user views in database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 73–84.

Devroye, L. 1986. *Non-Uniform Random Variate Generation*. Springer.

Dong, X. L., Halevy, A. Y., and Yu, C. 2009. Data integration with uncertainty. *VLDB J.*18, 2, 469–500.

FISHMAN, G. 1996. *Monte Carlo: Concepts, Algorithms, and Applications.* Springer.

FUHR, N. AND ROLLEKE, T. 1997. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM Trans. Inf. Syst. 15*, 1, 32–66.

GENTLE, J. E. 2003. *Random Number Generation and Monte Carlo Methods* 2nd Ed. Springer.

GETOOR, L. AND TASKAR, B., Eds. 2007. *Introduction to Statistical Relational Learning.* MIT Press.

GRIFFITHS, T. AND GHAHRAMANI, Z. 2005. Infinite latent feature models and the indian buffet process. In *Proceedings of the Conference on Advances in Neural Information Processing Systems (NIPS'05).*

GUHA, S. 2010. RHIPE – R and hadoop integrated processing environment. http://ml.stat.purdue.edu/rhipe/

GUPTA, R. AND SARAWAGI, S. 2006. Creating probabilistic databases from information extraction models. In *Proceedings of the International Conference on Very Large Databases (VLDB'06).* 965–976.

HENDERSON, S. G. AND NELSON, B. L., Eds. 2006. *Simulation*. North-Holland.

JAMPANI, R., XU, F., WU, M., PEREZ, L. L., JERMAINE, C. M., AND HAAS, P. J. 2008. MCDB: A Monte Carlo approach to managing uncertain data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 687–700.

KENNEDY, O. AND KOCH, C. 2010. PIP: A database system for great and small expectations. In *Proceedings of the International Conference on Data Engineering (ICDE'10).* 157–168.

KIMELFELD, B., KOSHAROVSKY, Y., AND SAGIV, Y. 2009. Query evaluation over probabilistic XML. *VLDB J. 18*, 5, 1117–1140.

KOCH, C. AND OLTEANU, D. 2008. Conditioning probabilistic databases. In *Proceedings of the International Conference on Very Large Databases (VLDB'08).*

LEHMANN, E. L. AND CASELLA, G. 1998. *Theory of Point Estimation* 2nd Ed. Springer.

MICHELAKIS, E., KRISHNAMURTHY, R., HAAS, P. J., AND VAITHYANATHAN, S. 2009. Uncertainty management in rule-based information extraction systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 101–114.

MILLER, JR., R. G. 1986. *Beyond ANOVA, Basics of Applied Statistics*. Wiley.

MURTHY, R. AND WIDOM, J. 2007. Making aggregation work in uncertain and probabilistic databases. In *Proceedings of the 1st International VLDB Workshop on Management of Uncertain Data (MUD'07).* 76–90.

NELSEN, R. B. 2006. *An Introduction to Copulas* 1st Ed. Springer Series in Statistics. Springer.

O'HAGAN, A. AND FORSTER, J. J. 2004. *Bayesian Inference* 2nd Ed. Volume 2B of Kendal l's Advanced Theory of Statistics. Arnold.

PANNETON, F., L'ECUYER, P., AND MATSUMOTO, M. 2006. Improved long-period generators based on linear recurrences modulo 2. *ACM Trans. Math. Softw. 32*, 1, 1–16.

PEREZ, L. L., ARUMUGAM, S., AND JERMAINE, C. M. 2010. Evaluation of probabilistic threshold queries in MCDB. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 687–698.

RE, C., DALVI, N. N., AND SUCIU, D. 2006. Query evaluation on probabilistic databases. *IEEE Data Engin. Bull. 29*, 1, 25–31.

RE, C., DALVI, N. N., AND SUCIU, D. 2007. Efficient top-k query evaluation on probabilistic data. In *Proceedings of the International Conference on Data Engineering (ICDE'07).* 886–895.

RE, C. AND SUCIU, D. 2008. Managing probabilistic data with MystiQ: The can-do, the could-do, and the can't-do. In *Proceedings of the SUM'08 Conference*. 5–18.

RE, C. AND SUCIU, D. 2009. The trichotomy of HAVING queries on a probabilistic database. *VLDB J. 18*, 5, 1091–1116.

ROBERT, C. P. AND CASELLA, G. 2004. *Monte Carlo Statistical Methods* 2nd Ed. Springer.

SEN, P., DESHPANDE, A., AND GETOOR, L. 2009. PrDB: Managing and exploiting rich correlations in probabilistic databases. *VLDB J. 18*, 5, 1065–1090.

SINGH, S., MAYFIELD, C., MITTAL, S., PRABHAKAR, S., HAMBRUSCH, S. E., AND SHAH, R. 2008. Orion 2.0: Native support for uncertain data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 1239–1242.

SRINIVASAN, A., CEPERLEY, D. M., AND MASCAGNI, M. 1997. Random number generators for parallel applications. In *Monte Carlo Methods in Chemical Physics*, Wiley, 13–36.

STONEBRAKER, M., BECLA, J., DEWITT, D. J., LIM, K.-T., MAIER, D., RATZESBERGER, O., AND ZDONIK, S. B. 2009. Requirements for science data bases and scidb. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR'09).* 26.

TAN, C. J. K. 2002. The PLFG parallel pseudo-random number generator. *Fut. Gener. Comput. Syst. 18*, 693–698.

TEH, Y., JORDAN, M., BEAL, M., AND BLEI, D. 2003. Hierarchical dirichlet processes. Tech. rep. 653, Department of Statistics, University of California, Berkeley.

THIAGARAJAN, A. AND MADDEN, S. 2008. Querying continuous functions in a database system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 791–804.

WANG, D. Z., MICHELAKIS, E., GAROFALAKIS, M., AND HELLERSTEIN, J. 2008a. BayesStore: Managing large, uncertain data repositories with probabilistic graphical models. In *Proceedings of the International Conference on Very Large Databases (VLDB'08).*

WANG, T.-Y., RE, C., AND SUCIU, D. 2008b. Implementing NOT EXISTS predicates over a probabilistic database. In *Proceedings of the MUD/QDB Conference*. 73–86.

XU, F., BEYER, K., ERCEGOVAC, V., HAAS, P. J., AND SHEKITA, E. J. 2009. E = M C 3: Managing uncertain enterprise data in a cluster-computing environment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 441–454.

Received July 2009; revised January 2011; accepted March 2011