MCDB: A Monte Carlo Approach to Managing Uncertain Data

Ravi Jampani[1] Fei Xu[1] Mingxi Wu[1]
Luis Leopoldo Perez[1] Christopher Jermaine[1] Peter J. Haas[2]

[1]University of Florida
Gainesville, FL, USA
{jampani,feixu,mwu,lperez,cjermain}@cise.ufl.edu

[2]IBM Almaden Research Center
San Jose, CA, USA
phaas@us.ibm.com

ABSTRACT

To deal with data uncertainty, existing probabilistic database systems augment tuples with attribute-level or tuple-level probability values, which are loaded into the database along with the data itself. This approach can severely limit the system's ability to gracefully handle complex or unforeseen types of uncertainty, and does not permit the uncertainty model to be dynamically parameterized according to the current state of the database. We introduce MCDB, a system for managing uncertain data that is based on a Monte Carlo approach. MCDB represents uncertainty via "VG functions," which are used to pseudorandomly generate realized values for uncertain attributes. VG functions can be parameterized on the results of SQL queries over "parameter tables" that are stored in the database, facilitating what-if analyses. By storing parameters, and not probabilities, and by estimating, rather than exactly computing, the probability distribution over possible query answers, MCDB avoids many of the limitations of prior systems. For example, MCDB can easily handle arbitrary joint probability distributions over discrete or continuous attributes, arbitrarily complex SQL queries, and arbitrary functionals of the query-result distribution such as means, variances, and quantiles. To achieve good performance, MCDB uses novel query processing techniques, executing a query plan exactly once, but over "tuple bundles" instead of ordinary tuples. Experiments indicate that our enhanced functionality can be obtained with acceptable overheads relative to traditional systems.

Categories and Subject Descriptors

H.2 [**Information Systems**]: Database Management

General Terms

Algorithms, Design, Languages, Performance

1. INTRODUCTION

The operation of virtually any modern enterprise requires risk assessment and decisionmaking in the presence of uncertain information. In the database research literature, the usual approach to addressing uncertainty employs an *extended relational model* (ERM), in which the classical relational model is augmented with attribute-level or tuple-level probability values, which are loaded into the database along with the data itself [1, 2, 4, 8, 11, 16, 19].

This ERM approach can be quite inflexible, however, for two key reasons. First, the representation of uncertainty is "hard wired" into the data model, and thus the types of uncertainty that can be processed are permanently limited by the specific model that has been chosen. If a new, unanticipated manifestation of uncertainty is later found to be important, but does not fit into the particular ERM being used, the only choice is to alter the data model itself. The user must then migrate the database to a new logical model, overhaul the database software, and likely change the physical database design.

Second, the uncertainty information, having been loaded in with the rest of the data, can be difficult to modify and limited in expressive power. Indeed, it rapidly becomes awkward to statically encode in an ERM anything more than the simplest types of uncertainty, such as (value, probability) pairs or standard distribution functions, e.g., in the form ("NormalDistn", meanVal, sigmaVal). If the probabilities associated with possible data values are derived from a complex statistical model, and the model or its parameters change, the probabilities typically need to be recomputed outside of the database and then loaded back in. It is therefore almost impossible to dynamically parameterize the uncertainty on the global state of the database or on results from arbitrary database queries.

As a result, there are many important types of uncertainty that seem difficult to handle in an ERM. An example is "extrapolation uncertainty," where the current state of the database is used to dynamically parameterize a statistical model that extrapolates the database into the past, the future, or into other possible worlds. Consider, for example, the TPC-H database schema.[1] We may wish to ask, "what would our profits have been last 12 months if we had raised all of our prices by 5%?" The problem is that we did *not* raise our prices by 5%, and so the relevant data are not present in the database. To handle this, we could use a Bayesian approach [28] that combines a "prior" distribution model of customer demand (having parameters that are derived from the entire database) with a customer's observed order size to create a "posterior" distribution for each customer's demand under the hypothetical price increase. After computing the posterior demand for each customer, we could check the new profits that would be expected; see Section 10, query Q4.

It is difficult to imagine implementing this analysis in an ERM. First, the statistical model is quite unique, so it is unlikely that it

[1]See www.tpc.org/tpch.

would be supported by any particular ERM. Moreover, the parameterization of the model depends upon the current database state in a complex way: in order to predict a customer's demand at a new price, it is necessary to consider the order sizes at the original price for all of the customers in the database and use this as input into a Bayesian statistical analysis. If the customer-demand analysis is to be performed on an ongoing basis, then it is necessary to parameterize the model on the fly. Finally, the posterior distribution function for a given customer's demand at the new price is quite complex; indeed, it cannot even be represented in closed form.

MCDB: The Monte Carlo Database System. In this paper, we propose a new approach to handling enterprise-data uncertainty, embodied in a prototype system called MCDB. MCDB does not encode uncertainty within the data model itself—all query processing is over the classical relational data model. Instead, MCDB allows a user to define arbitrary *variable generation* (VG) functions that embody the database uncertainty. MCDB then uses these functions to pseudorandomly generate realized values for the uncertain attributes, and runs queries over the realized values. In the "what if" profit scenario outlined above, the user could specify a VG function that, for a given customer, performs a Bayesian inference step to determine the posterior demand distribution for the customer at the new, discounted price, and then pseudorandomly generates a specific order quantity according to this distribution. Importantly, VG functions can be parameterized on the results of SQL queries over "parameter tables" that are stored in the database. By storing parameters rather than probabilities, it is easy to change the exact form of the uncertainty dynamically, according to the global state of the database. Such dynamic parameterization is highly desirable both for representing complex stochastic models of uncertainty, as described above, and for exploring the effect on a query result of different assumptions about the underlying data uncertainty.

Since VG functions can be arbitrary, it is very difficult to analytically compute the effect on the query result of the uncertainty that they embody. MCDB avoids this problem by, in effect, using the VG functions to generate a large number of independent and identically distributed (i.i.d.) realizations of the random database—also called "possible worlds"—on the fly, and running the query of interest over each of them. Using these Monte Carlo replicates, MCDB summarizes the effect of the underlying uncertainty in the form of an empirical probability distribution over the possible query results. Since MCDB relies on computational brute force rather than complicated analytics, it gracefully avoids common deficiencies of the various ERM approaches (see Section 2).

Our Contributions. The paper's contributions are as follows:

- We propose the first "pure" Monte Carlo approach toward managing uncertain data. Although others have suggested the possibility of Monte Carlo techniques in probabilistic databases [31], 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.
- We provide a syntax for specifying random tables that requires only a slight modification of SQL, and hence is easily understood by database programmers. The specification of VG functions 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 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 possible worlds. 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.

2. MONTE CARLO QUERY PROCESSING

VG functions provide a powerful and flexible framework for representing uncertainty, incorporating statistical methods directly into the database (similar in spirit to the MauveDB project [12]). One consequence of the extreme generality is that exact evaluation of query results—such as tuple appearance probabilities or the expected value of an aggregation query—is usually not feasible. From MCDB's point of view, a VG function is a "black box" with an invisible internal mechanism, and thus indirect means must be used to quantify the relationship between a VG function and the query results that it engenders. Specifically, MCDB invokes the VG functions to provide pseudorandom values, and then uses those values to produce and evaluate many different database instances ("possible worlds") in Monte Carlo fashion.

2.1 Monte Carlo Benefits

The need for Monte Carlo techniques is not necessarily a bad thing. Monte Carlo has several important benefits compared to the exact-computation approach that underlies virtually all existing proposals [1, 2, 4, 6, 7, 8, 10, 11, 33].

For example, unlike Monte Carlo, exact computation imposes strong restrictions both on the class of queries that can be handled and on the characteristics of the query answer that can be evaluated. Complex query constructs—e.g., `EXISTS` and `NOT IN` clauses, outer joins, or `DISTINCT` operators—cause significant difficulties for current exact approaches. Even relatively simple queries can result in #P complexity for query evaluation [11], and aggregation queries such as `SUM` and `AVG`, which are fundamental to OLAP and BI processing, pose significant challenges [25]. Moreover, it is often unclear how to compute important characteristics of the query output such as quantiles, which are essential for risk evaluation and decisionmaking. Of course, it is possible to extend the exact approach to handle broader classes of queries and inference problems, and work in this direction has been abundant [3, 6, 9, 23, 24, 29, 32, 35]. But adding more and more patches to the exact-computation approach is not a satisfactory solution: almost every significant extension to the approach requires new algorithms and new theory, making system implementation and maintenance difficult at best.

Another benefit of the Monte Carlo approach is that the same general-purpose methods apply to any correlated or uncorrelated uncertainty model. In contrast, general models for statistical correlation can be quite difficult to handle (and model) using exact computation. This is evidenced by the sheer number of approaches tried. Proposals have included: storing joint probabilities in an ERM, e.g., (`A1.value`, `A2.value`, probability) triplets to specify correlations between attributes [4], storing joint probabilities over small subsets of attributes [18, 33], and enhancing the stored probabilities with additional "lineage" information [1, 16]. Each of these models has its own sophisticated computational methods to measure the effect of the correlation—and yet none of them attempts to handle standard statistical dependencies such as those produced via a random walk (see Section 10, query Q3), much less dependencies described by complex models such as VARTA processes or copulas [5, 27]. Of course, one can always attempt to develop specialized algorithms to handle new types of correlation as they arise—but again, this is not a practical solution. At an abstract level, the task of computing probabilities based on many correlated input random variables can be viewed as equivalent to computing the value of an integral of a high-dimensional function. Such an integration task is extremely hard or impossible in the absence of very special structure; even the application of approximation methods, such as the central limit theorem, is decidedly nontrivial, since the pertinent random variables are, in general, non-identically distributed and dependent. Monte Carlo methods are well-known to be an effective tool for attacking this problem [15, 17].

Finally, Monte Carlo methods can easily deal with arbitrary, continuous distributions. It is possible to handle continuous distributions using the exact method, and relevant proposals exist [7, 10]. However, exact computation becomes difficult or impossible when continuous distributions do not have a closed-form representation; for example, evaluation of a "greater than" predicate requires expensive numerical integration. Such analytically intractable distributions arise often in practice, e.g., as posterior distributions in Bayesian analysis or as distributions that are built up from a set of base distributions by convolution and other operations.

2.2 Monte Carlo Challenges

Of course, the flexibility of the Monte Carlo approach is not without cost, and there are two natural concerns. First is the issue of performance. This *is* significant; implementation and performance are considered in detail in Sections 6 through 10 of the paper, where we develop our "tuple bundle" approach to query processing.

Second, MCDB merely estimates its output results. However, we feel that this concern is easily overstated. Widely accepted statistical methods can be used to easily determine the accuracy of inferences made using Monte Carlo methods; see Section 5. Perhaps more importantly, the probabilities that are stored in a probabilistic database are often very rough estimates, and it is unclear whether exact computation over rough estimates makes sense. Indeed, the "uncertainty" will often be expressed simply as a set of constraints on possible data values, with no accompanying probability values for the various possibilities. For example, the age of a customer might be known to lie in the set $\{35, 36, \ldots, 45\}$, but a precise probability distribution on the ages might be unavailable. In such cases, the user must make an educated guess about this probability distribution, e.g., the user might simply assume that each age is equally likely, or might propose a tentative probability distribution based on pertinent demographic data. As another example, probabilities for extraction of structured data from text are often based on approximate generative models, such as conditional random fields, whose parameters are learned from training data; even these already approximate probabilities are sometimes further approximated to facilitate storage in an ERM [19]. MCDB avoids allocating system resources to the somewhat dubious task of computing exact answers based on imprecise inputs, so that these resources can instead be used, more fruitfully, for sensitivity and what-if analyses.

3. SCHEMA SPECIFICATION

We now start to describe MCDB. As mentioned above, MCDB is based on possible-worlds semantics. A relation is *deterministic* if its realization is the same in all possible worlds, otherwise it is *random*. 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. We begin our description of MCDB by considering specification of random relations.

3.1 Schema Preliminaries

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. We follow [30] and assume that each random relation R can be viewed as a union of blocks of correlated tuples, where tuples in different blocks are independent. This assumption entails no loss of generality since, as an extreme case, all tuples in the table can belong to the same block. At the other extreme, a random relation made up of mutually independent tuples corresponds to the case in which each block contains at most one tuple.

3.2 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 above mean and standard deviation parameters for shifted blood pressure are stored in a single-row table `SPB_PARAM(MEAN, STD)` and that patient data are stored in a deterministic table `PATIENTS(PID, GENDER)`. Then the random table `SBP_DATA` can be specified as

```
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 `UNION`ed to create the realization of `SBP_DATA`. The `FOR EACH` clause specifies this outer loop. In general, every random `CREATE TABLE` specification 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 a relation or SQL expression (the relation `PATIENTS` in our example).

The standard library VG function `Normal` pseudorandomly generates independent and identically distributed (i.i.d.) 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, e.g., `p.PID` and `p.GENDER`.

3.3 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. Suppose that 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 (MONEY, LIVES_IN) value pairs for the different customers are conditionally mutually independent, given the REGION and SHAPE values 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

```
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. Finally, note that the parameters for the uncertainty model are stored in a space-efficient denormalized form; we emphasize that parameter tables are standard relational tables that can be indexed to boost processing efficiency.

3.4 Capturing ERM Functionality

As a variant of the above example, suppose that associated with each customer is a set of possible cities of residence, along with a probability for each city. Assuming that this information is stored in a table CITIES(CID, NAME, PROB), we change the definition of LIVES_IN to

```
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 [1, 4, 19].

Tuple-inclusion uncertainty as in [11] can also be represented within MCDB. Consider a variant of the example of Section 3.3 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 query 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
```

3.5 Structural Uncertainty

"Structural" uncertainty [18], i.e., fuzzy queries, can also be captured within the MCDB framework. For example, suppose that a table LOCATION(LID, NAME, CITY) describes customer locations, and another table SALES(SID, NAME, AMOUNT) contains 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 table:

```
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 overall result is now given by the query

```
SELECT l.CITY, SUM(s.AMOUNT)
FROM LOCATION l, SALES s, LS_JOIN j
WHERE l.TID = 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.

3.6 Correlated Attributes

Correlated attributes 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 attributes `VALUE1` and `VALUE2` corresponding to the generated values of `MONEY` and `LIVES_IN`.

3.7 Correlated Tuples

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, i.e., once per distinct value of `g`. For each group, the function returns a multi-row 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—corresponding to group `g`—in the generated table `SENSORS`.

4. 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. This process is described below.

4.1 Basic VG Function Interface

A VG function is implemented as a C++ class with 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 for random value generation.

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.

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 invokes the VG function's `Finalize()` method, which deletes any internal VG-function data structures.

4.2 Example VG Implementation

We illustrate the above ideas via a naive implementation of a very simple VG function, `DiscreteChoice` for strings. This VG function is slightly more general than the VG function defined in Section 3.3, in that the function accepts a set of character strings $x_1, x_2, \ldots, x_n$ and associated nonnegative "weights" $w_1, w_2, \ldots, w_n$, then normalizes the weights into a vector of probabilities $\mathcal{P} = (p_1, p_2, \ldots, p_n)$ with $p_i = w_i / \sum_j w_j$, and finally returns a random string X distributed according to $\mathcal{P}$, i.e., $P\{\, X = x_i \,\} = p_i$ for $1 \le i \le n$. The function uses a standard "inversion" method to generate the random string, which is based on the following fact. Let U be a random number uniformly distributed on $[0,1]$. Set $X = x_I$, where I is a random variable defined by $I = \min\{\, 1 \le i \le n \colon U < \sum_{j=1}^{i} p_j \,\}$. Then

$$P\{\, I = i \,\} = P\left\{ \sum_{j=1}^{i-1} p_j \le U < \sum_{j=1}^{i} p_j \right\} = p_i$$

for $1 \le i \le n$. That is, X is distributed according to $\mathcal{P}$.

This `DiscreteChoice` function has a single input table with two columns that contain the strings and the weights, respectively, so that each input parameter vector `v` to this function is of length 2; we denote these two entries as `v.str` and `v.wt`. The output table has a single row and column, which contains the selected string.

Our implementation is now as follows. The `Initialize()` method executes a statement of the form `myRandGen = new RandGen(seed)` to create and initialize a uniform pseudorandom-number generator `myRandGen` using the `seed` value that MCDB has passed to the method; a call to `myRandGen` returns a uniform pseudorandom number and, as a side effect, updates the

[2]A uniform pseudorandom number generator deterministically and recursively computes a sequence of seed values (typically 32 or 64 bit integers), which are then converted to floating-point numbers in the range $[0,1]$ by normalization. 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 [15] and [21, Ch. 3] for introductory and state-of-the-art discussions, respectively. The uniform pseudorandom numbers can then be transformed into pseudorandom numbers having the desired final distribution [13].

```
1   If newRep:
2      newRep = false
3      uniform = myRandGen()
4      probSum = i = 0
5      while (uniform >= probSum):
6         i = i + 1
7         probSum = probSum + (L[i].wt/totWeight)
8      return L[i].str
9   Else:
10     newRep = true
11     return NULL
```

Figure 1: The `OutputVals` method

value of `seed`. The method also allocates storage for a list `L` of parameter vectors; we can view `L` as an array indexed from 1. Next, the method initializes a class variable `totWeight` to 0; this variable will store the sum of the input weights. Finally, the method also sets a class variable `newRep` to `true`, indicating that we are starting a new Monte Carlo repetition (namely, the first such repetition). The `Finalize()` method de-allocates the storage for `L` and destroys `myRandGen`. The `TakeParams()` function simply adds the incoming parameter vector `v` to the list `L` and also increments `totWeight` by `v.wt`.

The most interesting of the methods is `OutputVals()`, whose pseudocode is given in Figure 1. When `OutputVals()` is called with `newRep = true` (line 1), so that we are starting a new Monte Carlo repetition, the algorithm uses inversion (lines 3–8) to randomly select a string from the list `L`, and sets `newRep` to `false`, indicating that the Monte Carlo repetition is underway. When `OutputVals()` is called with `newRep = false` (line 9), a Monte Carlo repetition has just finished. The method returns `NULL` and sets `newRep` to `true`, so that the method will correctly return a non-`NULL` value when it is next called.

5. INFERENCE AND ACCURACY

Using the `Inference` operator described in Section 8.4 below, MCDB returns its query results as a set of (t_i, f_i) pairs, where $t_1, t_2, \dots$ are the distinct tuples produced in the course of N Monte Carlo iterations and f_i is the fraction of the N possible worlds in which tuple t_i appears. Such results can be used to explore the underlying distribution of query answers in many different ways.

For example, in the presence of uncertain data, the answer X to an aggregation query Q such as `SELECT SUM(sales) FROM T`—where `T` is a random table—is no longer a fixed number, but a random variable, having a probability distribution that is unknown to the user. MCDB will, in effect, execute Q on N i.i.d. realizations of T, thereby generating N i.i.d. realizations of X. We can now plot the results in a histogram to get a feel for the shape of the distribution of X; see Section 10 for examples of such plots.

We can, however, go far beyond graphical displays: the power of MCDB lies in the fact that we can leverage over 50 years of Monte Carlo technology [17, 21] to make statistical inferences about the distribution of X, about interesting features of this distribution such as means and quantiles, and about the accuracy of the inferences themselves. For example, if we are interested in the expected value of the answer to Q, we can estimate $E[X]$ by $\bar{x}_N = N^{-1}\sum_{i=1}^{d} y_i n_i$, where $y_1, y_2, \dots, y_d$ are the distinct values of X produced in the course of the N Monte Carlo iterations, and n_i is the number of possible worlds in which $X = y_i$, so that $\sum_{i=1}^{d} n_i = N$. (In this example, the `SUM` query result is a single-row, single-attribute table, so that $y_i = t_i$ and $n_i = f_i N$.) We can also assess the accuracy of $\bar{x}_N$ as an estimator of $E[X]$: assuming N is large, the central limit theorem [34, Sec. 1.9] implies that, with probability approximately 95%, the quantity $\bar{x}_N$ estimates $E[X]$ to within $\pm 1.96\hat{\sigma}_N/\sqrt{N}$, where $\hat{\sigma}_N^2 = (N-1)^{-1}\sum_{i=1}^{d}(y_i - \bar{x}_N)^2 n_i$. If we obtain preliminary values of $\bar{x}_N$ and $\hat{\sigma}_N$, say, from a small pilot execution, then we can turn the above formula around and estimate the number of Monte Carlo replications needed to estimate $E[X]$ to within a desired precision; alternatively, we can potentially use a sequential estimation procedure as in [26] (this is a topic for future research).

Analogous results apply to estimation of quantiles [34, Sec. 2.6] and other statistics of interest. Indeed, we can use Kolmogorov's theorem [34, p. 62] to approximate the entire cumulative distribution function of X. For example, denoting this function by F and the empirical distribution function by F_N, Kolmogorov's theorem implies that with probability approximately 95%, the absolute difference $|F(x) - F_N(x)|$ is bounded above by $1.36/\sqrt{N}$ for all x. If the distribution of X is known to have a probability density function, then this function can be estimated using a variety of techniques [14]; note that a histogram can be viewed as one type of density estimator. Besides estimation, we can perform statistical tests of hypotheses such as "the expected value of the result of Q_1 is greater than the expected value of the result of Q_2." If Q_1 and Q_2 correspond to two different business policies, then we are essentially selecting the best policy, taking into account the uncertainty in the data; more sophisticated "ranking and selection" procedures can potentially be used with MCDB [21, Ch. 17].

More generally, the answer X to a query can be an entire (random) table. In this case, we can, for example, use the results from MCDB to estimate the true probability that a given tuple t_i appears in the query answer; this estimate is simply f_i. We can also compute error estimates on f_i, perform hypothesis tests on appearance probabilities, and so forth. The idea is to consider a transformation $\phi_i(X)$ of the random, table-valued query result X, where $\phi_i(X) = 1$ if t_i appears in X, and $\phi_i(X) = 0$ otherwise. Then, on each possible world, the result of our transformed query is simply a number (0 or 1), and the previous discussion applies in full generality, with $f_i = \bar{x}_N$.

In summary, MCDB permits the use of powerful inference tools that can be used to study results of queries on uncertain data. Many other estimation methods, stochastic optimization techniques, hypothesis tests, and efficiency-improvement tricks are potentially applicable within MCDB, but a complete discussion is beyond the scope of this paper.

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

Logically, 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. 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 answer sets have been generated, all of the output tuples are then merged into a single file, sorted, and scanned to determine the number of iterations 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 runs each query one time, regardless of N. In MCDB, Q is evaluated only *once*, whatever value of N is supplied by the user. Each "database tuple" that 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 database instance.

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.

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 above—and processed by a relational operator, MCDB permits this value to be discarded and then later re-materialized 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. Tuple bundles are manipulated using the new operators described in Section 8 and the modified versions of classical relational operators described in Section 9. In general, there are many possible ways in which the realized attribute values for a random table R can be bundled. The only requirement on a set of tuple bundles $t_1, t_2, \ldots, t_k$ is that, for each i, the set $r_i = \bigcup_j t_j[i]$ corresponds precisely to the ith realization of R.

There are many possible ways to bundle individual tuples together across Monte Carlo database instances. 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 3.2, 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 table, MCDB creates one tuple bundle for every distinct combination of constant-attribute values encountered. MCDB often stores values for non-constant attributes in a highly compressed form by storing only the seed used to pseudorandomly generate the values, rather than an actual array of values.

A tuple bundle t in MCDB may have a special random attribute called the *isPresent* attribute. The value of this attribute for the ith iteration 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 Monte Carlo 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 example, 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 move through the query plan. We begin by describing in some detail the operations unique to MCDB.

8.1 The Seed Operator

For a given random table R and VG function V, the `Seed` operator appends to each tuple created by R's `FOR EACH` statement an integer unique to the (tuple, VG function) pair. This integer serves as the pseudorandom seed for V when expanding the tuple into an uncompressed tuple bundle.

8.2 The Instantiate Operator

The `Instantiate` operator is perhaps the most unique and fundamental operator used by MCDB. For a random table R, this operator uses a VG function to generate a set of attribute values—corresponding to a Monte Carlo iteration—which is appended to the individual tuple bundles in R. To understand the workings of `Instantiate`, it is useful to consider a slightly modified version of the example in Section 3.2, in which the mean and variance for the shifted blood pressure reading explicitly depend on a patient's gender, so that the table `SPB_PARAM` now has two rows and an additional `GENDER` attribute.

```
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
    WHERE s.GENDER = p.GENDER))
  SELECT p.PID, p.GENDER, b.VALUE
  FROM SBP b
```

The `Instantiate` operator accepts the following seven parameters, which are extracted from R's `CREATE TABLE` statement:

- Q_{out}. This is the answer set for the "outer" query that is the source for the tuples in the `FOR EACH` clause. In our example, Q_{out} is simply the result of a table scan over the relation `PATIENTS`. However, as in Sections 3.5 and 3.7, Q_{out} may also be the result of a query. In general, a random relation R may be defined in terms of multiple VG functions, in which case R is constructed via a series of invocations of the `Instantiate` operation, one for each VG function.

- VG. This is the variable generation function that will be used to generate attribute values.

- $VGAtts$. This is the set of attributes whose values are produced by the VG function and are to be used to update the tuple bundles. In our example, $VGAtts$ comprises the single attribute `Normal.VALUE`.

- $OutAtts$. This is the set of attributes from Q_{out} that should appear in the result of `Instantiate`. In our example, $OutAtts$ comprises the attributes `p.PID` and `p.GENDER`.

- $Q_{in,1}, Q_{in,2}, \ldots, Q_{in,r}$. These are the answer sets for the "inner" input queries used to supply parameters to VG. In our example, there is only one inner input query, and so $Q_{in,1}$ is the result of `SELECT s.MEAN, s.STD, s.GENDER FROM SBP_PARAM s`. Note that the attribute `s.GENDER` is required because this attribute will be used to join Q_{out} with $Q_{in,1}$.

- $InAtts_1, InAtts_2, \ldots, InAtts_r$. Here $InAtts_i$ is the set of those attributes from the ith inner query that will be fed into VG. In our example, $InAtts_1$ consists of `s.MEAN` and `s.STD`.

- $B_1, B_2, \ldots, B_r$. Here B_i is the boolean join condition that links the ith inner query to the outer query. In our example, B_1 is the predicate "`s.GENDER = p.GENDER`".

We first assume (as in our example) that there is only one inner query, so that we have only Q_{in}, $InAtts$, and B in addition to Q_{out}, $VGAtts$, and $OutAtts$; extensions to multiple inner queries (and multiple VG functions) are given below. Given this set of arguments, an outline of the steps implemented by the `Instantiate` operator to add random attribute values to a stream of input tuples is as follows. The process is illustrated in Figure 2.

1. First, the input pipe supplying tuples from Q_{out} is forked, and copies of the tuples from Q_{out} are sent in two "directions". One fork bypasses the VG function entirely, and is used only to supply values for the attributes specified in $OutAtts$. For this particular fork, all of the attributes present in Q_{out} except for those in $OutAtts \cup \{seed\}$ are projected away and then all of the result tuples are sorted based upon the value of the tuple's seed.

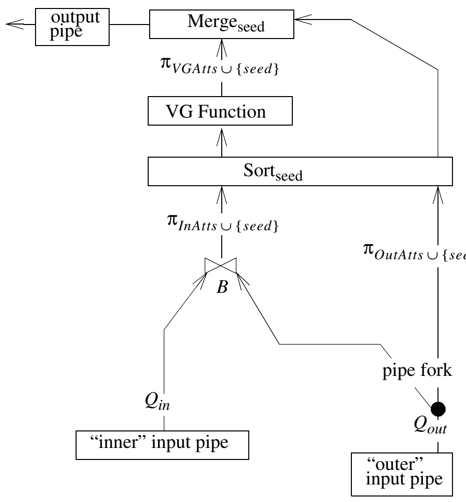

Figure 2: The `Instantiate` operation for a single inner input query.

```
1   $VG$.Initialize($t_i.seed$)
2   For each tuple $s$ in the group $S_i$:
3       $VG$.TakeParams($\pi_{InAtts}(s)$)
4   $OutputTuples = \langle\rangle$
5   For $j = 1$ to $N$:
6       For $k = 1$ to $\infty$:
7           $temp = VG$.OutputVals()
8           If $temp$ is NULL, then break
9           $OutputTuples[j][k] = \pi_{VGAtts}(temp) \bullet t_i.seed$
10  $VG$.Finalize()
```

Figure 3: Step four of the `Instantiate` operator.

2. The second fork is used to supply parameters to the VG function. Using this fork, the set $S = Q_{out} \bowtie_B Q_{in}$ is computed; all attributes except for the VG function seed and the attributes in $InAtts$ are then projected away after the join.

3. Next, S is grouped (ordered) so that if two tuples s_1, s_2 in S were produced by the same $t \in Q_{out}$, then s_1 and s_2 are always found in the same group. This is easily accomplished by sorting S on the seed value contained in each tuple. Note that tuples in the same group have the same seed value.

4. Then, for each group S_i in S, the VG function produces a result array $OutputTuples$ using the pseudocode in Figure 3. After the pseudocode is completed for a given S_i, the rows in $OutputTuples$ are sent onwards, to update the tuple bundles. In the figure, $t_i \in Q_{out}$ is the outer tuple corresponding to S_i, $t_i.seed$ is the common seed value, and $\bullet$ denotes tuple concatenation. This code first feeds each of the parameter values in the set S_i into the function VG (line 3). The code then performs N Monte Carlo iterations (lines 5–9). The seed value $t_i.seed$ that produced the set of tuple bundles is appended to the row, so that it is possible to identify which tuple from the outer input query was used to produce the row.

5. Finally, the results of steps 1 and 4 are merged (joined) based upon the seed values, so that the attributes supplied by Q_{out} can be combined with the attributes produced by the VG function. During this merge step, the putative instantiated row of R that has just been created may be filtered out by applying the final `WHERE` predicate, if any, that appears after the final `SELECT` clause in the `CREATE TABLE` statement.

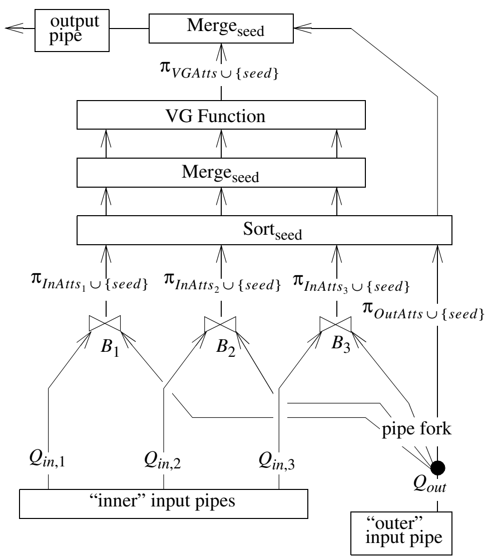
output pipe
$\text{Merge}_{\text{seed}}$
$\pi_{VGAtts \cup \{seed\}}$
VG Function
$\text{Merge}_{\text{seed}}$
$\text{Sort}_{\text{seed}}$
$\pi_{InAtts_1 \cup \{seed\}}$
$\pi_{InAtts_2 \cup \{seed\}}$
$\pi_{InAtts_3 \cup \{seed\}}$
$\pi_{OutAtts \cup \{seed\}}$
B_1
B_2
B_3
pipe fork
$Q_{in,1}$
$Q_{in,2}$
$Q_{in,3}$
Q_{out}
"inner" input pipes
"outer" input pipe

Figure 4: The `Instantiate` operation for multiple inner input queries.

Handling multiple inner queries. When there are multiple inner queries that supply input parameters to the VG function, the foregoing process must be generalized slightly. The generalization is pictured in Figure 4. Rather than only forking the outer input pipe that supplies tuples from Q_{out} in two directions, one additional fork is required for each additional inner query. Each of the resulting parameter streams is merged or grouped so that each group contains only parameters with exactly the same seed value. Once this single set of parameters is obtained, it is sent to the VG function via calls to `TakeParams`, and the rest of the `Instantiate` operation proceeds exactly as described above.

Handling multiple VG functions. When k (> 1) VG functions appear in the same `CREATE TABLE` statement, `Instantiate` is not changed at all; instead, k `Instantiate` operations are executed, and then a final join is used to link them all together. In more detail, MCDB first seeds each outer tuple with k seeds, one for each VG function, and then appends a unique synthetic identifier to the tuple. The resulting stream of tuples is then forked k ways. The kth fork is sent into an `Instantiate` operation for the kth VG function, essentially implementing a modified `CREATE TABLE` statement in which all references to VG functions other than the kth have been removed and in which the synthetic identifier is added to the final `SELECT` list. MCDB executes a k-way join over the k result streams, using the synthetic identifiers as the join attributes (and appropriately projecting away redundant attributes).

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 non-constant attribute. This is problematic when implementing an operation such as relational join, which typically requires ordering the input tuples by their join attributes via sorting or hashing.

In such a situation, it is necessary to apply the `Split` 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 non-constant. 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 = `(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 $Atts = \{\text{age}\}$ yields two tuple bundles t_1 = `(Jane, Smith, 20, (T, F, T, F))` and t_2 = `(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, unbundled tuples, where unbundled 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. (Typically, one attribute of t' will be a primary key, so that t' will appear at most once per Monte Carlo iteration.) Note that f estimates p, the true probability that t' will appear in a realization of the query result.

MCDB implements `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 of i values for which $t[i].isPres$ = `true`. Let this value be n. The operator then outputs a tuple with attribute value $t[\cdot].att$ for each $att \in Atts$, together with the relative frequency $f = n/N$.

9. STANDARD RELATIONAL OPS

In addition to the new operations described above, 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, $t[i].isPres = B(t[i]) \wedge t[i].isPres$. 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, $t[i].isPres$ = `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 `Selection` operation can be executed in $O(1)$ time by simply accepting or rejecting the entire tuple bundle based on the unique value of each of these attributes.

9.2 Projection

Projection in MCDB is nearly identical to projection in a classical system, with a few additional considerations. If a non-constant 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 re-generated, projection of an attribute does not necessarily remove the seed for that attribute unless this is explicitly requested.

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 as before, 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 above, 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 non-constant 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 non-constant 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 any two bundles in the same group have the 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. Standard SQL semantics apply, so that if the foregoing sum is empty for some value of i, then $t[i].agg = \texttt{NULL}$. Other aggregation functions are implemented similarly.

10. EXPERIMENTS

The technical material in this paper has focused upon the basic Monte Carlo framework employed by MCDB, upon the VG function interface, and upon MCDB's implementation details. Our experimental study is similarly focused, and has two goals:

1. To demonstrate examples of non-trivial, "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?

There are many possible novel, interesting, and rich examples to study. Given our space constraints, we choose to focus on four such examples in depth, to give a better feel both for the novel applications amenable to MCDB and for the performance of our initial prototype.

Basic Experimental Setup. We generate a 20GB instance of the TPC-H database using TPC-H's `dbgen` program and use MCDB to run four non-trivial "what-if" aggregation queries over the generated database instance. Each of the four 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 multi-threaded prototype has full support for the VG function interface described in the paper, 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, as 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 four queries are run on a dedicated and relatively low-end, $3000 server machine with four, 160GB ATA hard disks and eight, 2.0GHz cores partitioned over two CPUs. The system has eight GB of RAM and runs the Ubuntu distribution of the Linux OS.

Queries Tested. The four benchmark queries we study are each computationally expensive, involving joins of large tables, expensive VG-function evaluation, 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 (assumed to be 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. Once 1996 is generated, the additional revenue is computed.

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 for each component of each order, the maximum duration is selected.

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 have a mean of -0.02

Query	1 iter	10 iters	100 iters	1000 iters
Q1	25 min	25 min	25 min	28 min
Q2	36 min	35 min	36 min	36 min
Q3	37 min	42 min	87 min	222 min[a]
Q4	42 min	45 min	60 min	214 min

[a]Measurement based on 350 Monte Carlo iterations

Figure 5: Wall-clock running times.

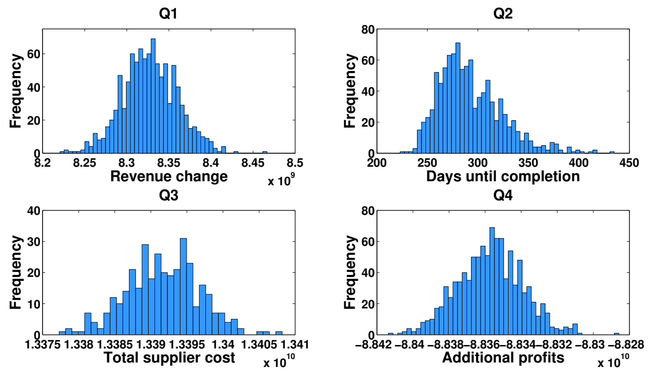

Figure 6: Empirical distributions for answers to Q1–Q4.

and a variance of 0.04. The most inexpensive price available for each part is then used to compute the total supplier cost.

Query Q4. This query is the one mentioned in Section 1, which estimates the effect of a 5% customer price increase on an organization's profits. The Bayesian VG function used in this query to predict a customer's demand at a new price appears impossible to integrate, 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 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 resulting demand d^*.

The inner workings of the VG function are described in more detail in the Appendix, but to make use of this VG function we must first issue a query to parameterize the prior version of D_p, and then for each customer, we feed the actual values p^* and d^* as well as the proposed price increase to the VG function, which then "guesses" the new demand. This new demand is then used to calculate the change in profit.

Results. The results obtained by running the four queries are given above in Figures 5 and 6. To put the running times in perspective, we ran a foreign key join over `partsupp`, `lineitem`, and `orders` in the Postgres DBMS, 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 relational query processing engine.

Figure 6 plots a histogram for all observed aggregate values over the four queries. The 1000 i.i.d. Monte Carlo samples obtained for each query do an excellent job of accurately summarizing the true distribution of aggregate values. For example, for query Q1, the inferred mean aggregate value is 8.3277e+09; 95%, central-limit-theorem-based bounds on this value (see Section 5) show an error of only $\pm 0.02\%$.

Remarkably, we found that for the first two queries, the number of Monte Carlo iterations had no effect on the running time. For query Q1, the naive approach of simply running the query 1000 times to complete 1000 Monte Carlo iterations would take over 400 hours to complete, whereas the MCDB approach takes 28 minutes. This illustrates very clearly the benefit of MCDB's tuple bundle approach to query processing, where the query is run only once and bundles of Monte Carlo values are stored within each tuple. Even for a large database, much of the cost in a modern database system is related to performing in-memory sorts and hashes, and these costs tend to be constant no matter how many Monte Carlo iterations are employed by MCDB.

In queries Q3 and Q4, MCDB was somewhat more sensitive to the number of Monte Carlo iterations, though even for the "worst" query (Q3), the MCDB time for 350 iterations was only six times that for a single iteration. The reason for the relatively strong influence of the number of Monte Carlo iterations on Q3's running time is that this query produces twelve individual, correlated tuple bundles for each and every tuple in `partsupp`, which results in 96 million large tuple bundles being produced by the VG function, where bundle size is proportional to the number of Monte Carlo iterations. Because of the sort-based `GROUP BY` operations in the query, the materialized attribute values needed to be carried along through most of the query processing, and had to be stored on disk. For 1000 Monte Carlo iterations, the resulting terabyte-sized random relation exceeded the capabilities of our benchmarking hardware, and so our observation of 222 minutes was obtained using a value of 350 iterations. We conjecture that replacing sort-based joins and grouping operations with hash-based operations will go a long way towards alleviating such difficulties.

Query Q4's sensitivity to the number of Monte Carlo iterations is related to its very expensive Bayesian VG function. For 1000 iterations, this function's costly `OuputVals` method is invoked nearly ten *billion* times, and this cost begins to dominate the query execution time. The cost of the VG function is made even more significant because our initial attempt at parallelizing the `Instantiate` implementation was somewhat ineffective, and MCDB had a very difficult time making use of all eight CPU cores available on the benchmarking hardware. We suspect that future research specifically aimed at `Instantiate` could facilitate significant speedups on such a query. Even so, the 214 minutes required by MCDB to perform 1000 trials is only 0.5% of the 700 hours that would be required to naively run the query 1000 times.

Although the TPC-H database generated by `dbgen` is synthetic, some of the qualitative results shown in Figure 6 are still interesting. In particular, we point to Q2, where MCDB uncovers evidence of a significant, long tail in the distribution of anticipated times until all existing orders are complete. If this were real data, the tail would be indicative of a significant need to be more careful in controlling the underlying order fulfillment process!

11. CONCLUSIONS

This paper describes an initial attempt to design and prototype a Monte Carlo-based system for managing uncertain 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 issues we intend to explore in future work include:

- *Query optimization.* The problem of costing alternative query plans appears to be challenging, as does the possibility of using query feedback to improve the optimizer. A related issue is to automatically detect when queries can be processed exactly and very efficiently, and have the MCDB system respond accordingly; the idea would be to combine our Monte Carlo approach with existing exact approaches in the literature, in an effective manner. We also plan—in the spirit of [32, 35]—to combine MCDB's processing methods with classical DBMS technology such as indexing and pre-aggregation, to further enhance performance.

- *Error control.* In our current prototype, the user must specify the desired number of Monte Carlo iterations, which can be hard to do without guidance. Our goal is to have the user specify precision and/or time requirements, and have the system automatically determine the number of iterations. Alternatively, it may be desirable to have the system return results in an online manner, so that the user can decide on the fly when to terminate processing [20, 22]. As indicated in Section 5, there is a large amount of existing technology that can potentially be leveraged here. Closely related to this issue is the question of how to define an appropriate syntax for specifying the functionals of the query-output distribution required by the user, along with the speed and precision requirements. Finally, we hope to exploit knowledge of these requirements to tailor MCDB's processing methods individually for each query, thereby improving efficiency. The functionality discussed in [24] is also of interest in this regard.

- *Improved risk assessment.* For purposes of risk assessment, we often want to estimate quantiles of the distribution of a query result. This task can be challenging for extreme quantiles that correspond to rare events. We hope to leverage Monte Carlo techniques, such as importance sampling [21, Ch. 11], that are known to be effective for such problems. Importance sampling can also potentially be used to "push down" selections into the VG function, i.e., to only generate sample tuples that satisfy selection predicates; see [36].

- *Correlated relations.* We are currently investigating the best way to handle correlation between random relations. One approach—which can be handled by the current prototype but may not be the most efficient possible—is to denormalize the random tables as necessary; that is, to ensure that any correlated attributes appear jointly in the same table. Other possible approaches include allowing a random relation R to appear in the specification of another random relation table S, and to allow VG functions to return a set of output tables.

- *Lineage.* We also note that our system does not explicitly track data lineage (also called provenance) as does a system like Trio [1]. It may be possible, however, to combine our Monte Carlo methods with lineage-management technology.

- *Non-relational applications.* Finally, we hope to extend the techniques and ideas developed here to other types of data, such as uncertain XML [23], as well as to other types of data-processing environments.

Overall, the approach embodied in MCDB has the potential to facilitate real-world risk assessment and decisionmaking under data uncertainty, both key tasks in a modern enterprise.

12. ACKNOWLEDGMENTS

We wish to thank the anonymous referees for several comments and suggestions that have improved the paper. Material in this paper is based upon work supported by the National Science Foundation under grant no. 0347408 and grant no. 0612170.

13. REFERENCES

[1] P. Agrawal, O. Benjelloun, A. D. Sarma, C. Hayworth, S. U. Nabar, T. Sugihara, and J. Widom. Trio: A system for data, uncertainty, and lineage. In *VLDB*, 2006.

[2] P. Andritsos, A. Fuxman, and R. J. Miller. Clean answers over dirty databases: A probabilistic approach. In *ICDE*, page 30, 2006.

[3] L. Antova, C. Koch, and D. Olteanu. 10^{10^6} worlds and beyond: Efficient representation and processing of incomplete information. In *ICDE*, pages 606–615, 2007.

[4] L. Antova, C. Koch, and D. Olteanu. MayBMS: Managing incomplete information with probabilistic world-set decompositions. In *ICDE*, pages 1479–1480, 2007.

[5] B. Biller and B. L. Nelson. Modeling and generating multivariate time-series input processes using a vector autoregressive technique. *ACM Trans. Modeling Comput. Simulation*, 13(3):211–237, 2003.

[6] D. Burdick, A. Doan, R. Ramakrishnan, and S. Vaithyanathan. OLAP over imprecise data with domain constraints. In *VLDB*, pages 39–50, 2007.

[7] R. Cheng, D. V. Kalashnikov, and S. Prabhakar. Evaluation of probabilistic queries over imprecise data in constantly-evolving environments. *Inf. Syst.*, 32(1):104–130, 2007.

[8] R. Cheng, S. Singh, and S. Prabhakar. U-DBMS: A database system for managing constantly-evolving data. In *VLDB*, pages 1271–1274, 2005.

[9] R. Cheng, S. Singh, S. Prabhakar, R. Shah, J. S. Vitter, and Y. Xia. Efficient join processing over uncertain data. In *CIKM*, pages 738–747, 2006.

[10] D. Chu, A. Deshpande, J. M. Hellerstein, and W. Hong. Approximate data collection in sensor networks using probabilistic models. In *ICDE*, page 48, 2006.

[11] N. N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *VLDB J.*, 16(4):523–544, 2007.

[12] A. Deshpande and S. Madden. MauveDB: supporting model-based user views in database systems. In *SIGMOD*, pages 73–84, 2006.

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

[14] L. Devroye and G. Lugosi. *Combinatorial Methods in Density Estimation*. Springer, 2001.

[15] G. Fishman. *Monte Carlo: Concepts, Algorithms, and Applications*. Springer, 1996.

[16] N. Fuhr and T. Rölleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM Trans. Inf. Syst.*, 15(1):32–66, 1997.

[17] J. E. Gentle. *Random Number Generation and Monte Carlo Methods*. Springer, second edition, 2003.

[18] L. Getoor and B. Taskar, editors. *Introduction to Statistical Relational Learning*. MIT Press, 2007.

[19] R. Gupta and S. Sarawagi. Creating probabilistic databases from information extraction models. In *VLDB*, pages 965–976, 2006.

[20] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD*, pages 171–182, 1997.

[21] S. G. Henderson and B. L. Nelson, editors. *Simulation*. North-Holland, 2006.

[22] C. M. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable approximate query processing with the DBO engine. In *SIGMOD*, pages 725–736, 2007.

[23] B. Kimelfeld and Y. Sagiv. Matching twigs in probabilistic XML. In *VLDB*, pages 27–38, 2007.

[24] B. Kimelfeld and Y. Sagiv. Maximally joining probabilistic data. In *PODS*, pages 303–312, 2007.

[25] R. Murthy and J. Widom. Making aggregation work in uncertain and probabilistic databases. In *Proc. 1st Int. VLDB Work. Mgmt. Uncertain Data (MUD)*, pages 76–90, 2007.

[26] A. Nadas. An extension of a theorem by Chow and Robbins on sequential confidence intervals for the mean. *Ann. Math. Statist.*, 40(2):667–671, 1969.

[27] R. B. Nelsen. *An Introduction to Copulas*. Springer, second edition, 2006.

[28] A. O'Hagan and J. J. Forster. *Bayesian Inference*. Volume 2B of *Kendall's Advanced Theory of Statistics*. Arnold, second edition, 2004.

[29] J. Pei, B. Jiang, X. Lin, and Y. Yuan. Probabilistic skylines on uncertain data. In *VLDB*, pages 15–26, 2007.

[30] C. Re, N. N. Dalvi, and D. Suciu. Query evaluation on probabilistic databases. *IEEE Data Eng. Bull.*, 29(1):25–31, 2006.

[31] C. Re, N. N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *ICDE*, pages 886–895, 2007.

[32] C. Re and D. Suciu. Materialized views in probabilistic databases for information exchange and query optimization. In *VLDB*, pages 51–62, 2007.

[33] P. Sen and A. Deshpande. Representing and querying correlated tuples in probabilistic databases. In *ICDE*, pages 596–605, 2007.

[34] R. J. Serfling. *Approximation Theorems of Mathematical Statistics*. Wiley, 1980.

[35] S. Singh, C. Mayfield, S. Prabhakar, R. Shah, and S. E. Hambrusch. Indexing uncertain categorical data. In *ICDE*, pages 616–625, 2007.

[36] J. Xie, J. Yang, Y. Chen, H. Wang, and P. Yu. A sampling-based approach to information recovery. In *ICDE*, 2008. To appear.

APPENDIX

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 increase_per_cust, 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

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 prc_hist, 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)
```

Details of VG function for query Q4. We define the prior distribution indirectly, in terms of the stochastic mechanism used to generate a realization of D_p. This mechanism works by generating random variables p_0 and d_0 according to independent gamma distributions Gamma(k_p, θ_p) and Gamma(k_d, θ_d), and then setting $D_p = (d_0/p_0)(p_0 - p)$. Here the shape parameters are $k_p = k_d =$ 2.0, and the scale parameters are $\theta_p = \frac{4}{3}\times$ (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 of actual transactions involving the part.

One way of viewing this process is that we have defined a probability distribution over the space of linear demand curves; i.e., p_0 is the price at which the customer will purchase nothing, and d_0 is the customer's demand if the price offered is 0. 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. We generate a random demand D_p by first generating a random demand function and then evaluating this function at the price of interest.

Given the observation (p^*, d^*) for a customer, the next task is to determine the customer's posterior demand distribution by first determining the posterior distribution of the customer's entire demand function. Roughly speaking, we define the posterior probability density function over the space of linear demand functions to be the prior density over this space, conditioned on the observation that the function intersects the point (p^*, d^*); we can write down an expression for the posterior density, up to a normalization factor, using Bayes' rule. Although we cannot compute the normalizing constant—and hence the demand-function density—in closed form, we can generate random demand functions 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 sampling algorithm to generate a demand function, and then (3) evaluating this function at the point $1.05p^*$.

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 and d_0 is $f_{p_0,d_0}(x, y) = g_p(x)g_d(y)$. If a demand curve passes through the point (d^*, p^*), 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

$$\begin{aligned}
&P\{\, d_0 = x, p_0 = y \mid p_0 = p^*d_0/(d_0 - d^*)\,\}\\
&\quad\propto P\{\, p_0 = p^*d_0/(d_0 - d^*) \mid d_0 = x, p_0 = y\,\}\\
&\qquad\times P\{\, d_0 = x, p_0 = y\,\}\\
&\quad= h(x, y)g_p(x)g_d(y)\\
&\quad= h(x, y)g_p(x)g_p\big(p^*x/(x - d^*)\big).
\end{aligned}$$

That is, $h_d(x) = cg_d(x)g_p\big(p^*x/(x - d^*)\big)$ 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 is easy to verify, and the proof is standard [13]. 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, we compute the customer's demand at the new price by $D = (d_0/p_0)(p_0 - 1.05p^*)$.

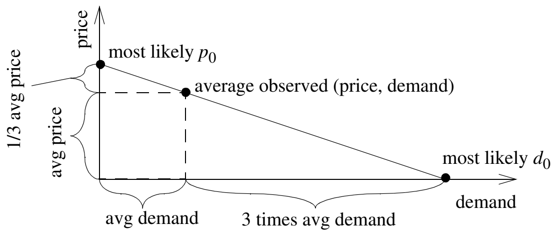

Figure 7: Most likely demand curve under prior distribution.