E = MC^3: Managing Uncertain Enterprise Data in a Cluster-Computing Environment

Fei Xu* Kevin Beyer† Vuk Ercegovac† Peter J. Haas† Eugene J. Shekita†

* University of Florida
Gainesville, FL, USA
feixu@cise.ufl.edu

† IBM Almaden Research Center
San Jose, CA, USA
{kbeyer,vercego,phaas,shekita}@us.ibm.com

ABSTRACT

Modern enterprises must manage uncertain data for purposes of risk assessment and decisionmaking under uncertainty. The Monte Carlo approach embodied in the MCDB system of Jampani et al. is well suited for such a task. MCDB can support industrial strength business-intelligence queries over uncertain warehouse data. Moreover, MCDB's extensible approach to specifying uncertainty can also capture complex stochastic prediction models, allowing sophisticated "what-if" analyses within the DBMS. The MCDB computations can be highly CPU intensive, but offer the potential for massive parallelization. To realize this potential, we provide a new system, called MC^3 (Monte Carlo Computation on a Cluster), that extends the MCDB approach to the map-reduce processing framework. MC^3 can exploit the robustness and scalability of map-reduce, and can handle data stored in non-relational formats. We show how MCDB query plans over "tuple bundles" can be translated to sequences of map-reduce operations over nested data, and describe different parallelization schemes. We also provide and analyze several novel distributed algorithms for adding pseudorandom number seeds to tuple bundles. These algorithms ensure statistical correctness of the Monte-Carlo computations while minimizing the seed length. Our experiments show that MC^3 can scale well for a variety of workloads.

Categories and Subject Descriptors

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

General Terms

Algorithms, Design, Languages, Performance

Keywords

Uncertain Data, Map-Reduce, Monte Carlo, JSON, JAQL

SIGMOD'09, June 29–July 2, 2009, Providence, Rhode Island, USA.

1. INTRODUCTION

There is an increasing need for tools that facilitate enterprise risk assessment and decision making in the face of uncertain data. The problem of data uncertainty is becoming acute, due to the increasing use of data integration, automated information extraction, and data anonymization for privacy protection, as well as the growing prevalence of RFID and sensor data. Database researchers have therefore developed a variety of prototype systems [1, 2, 3, 16, 26, 28, 32] for managing uncertain data. Among these prototypes, the Monte Carlo relational Database System (MCDB) [16] seems especially promising for general decision-support applications.

MCDB's Monte Carlo approach permits processing of industrial strength Business Intelligence (BI) queries — e.g., complex SQL aggregation queries — over warehouses where the data is uncertain and has complex statistical dependencies between attributes and tuples. Perhaps more importantly, the MCDB uncertainty model is completely extensible: uncertainty is specified via user-defined Variable Generation (VG) functions, which are used to pseudorandomly generate realized values for uncertain attributes. As a consequence, the MCDB model subsumes the uncertainty models used in current prototype systems. For example, MCDB can capture discrete models of uncertainty such as the attribute-value and tuple-inclusion uncertainty models used in systems such as MayBMS, Trio, and Mystiq [1, 2, 3]. Moreover, MCDB is well suited to a very important class of scenarios in which uncertainty arises due to the need to extrapolate missing data using probabilistic models, as is often the case with financial, banking, marketing, fraud-detection, and decision-support applications. In this latter setting, MCDB permits sophisticated, data-intensive stochastic modeling and prediction without the need to continually move data back and forth between the DBMS and a statistical or simulation package such as R or ARENA. Thus the user can assess not only the uncertainty in the results of BI queries over uncertain data, but can also ask what-if questions such as "What will be the mean effect on my profits next quarter if I increase my prices by 5%?" or "What is the probability that the average value of my New York customers' portfolios will drop by more than 10% over the next month?" To achieve the foregoing new functionality without unacceptably increasing processing overhead, MCDB uses a novel processing technique in which a query plan is executed exactly once, but over "tuple bundles" rather than ordinary

tuples. A tuple bundle represents the values of a tuple over all Monte Carlo replications — equivalently, over all sampled "possible worlds" — see Section 2.1.

In this paper, we provide a new system, called MC^3 (Monte Carlo Computation on a Cluster), that extends the MCDB approach to a map-reduce framework. Our motivation for this work is threefold:

- As the amount of data continues to increase exponentially, massively-parallel processing techniques are becoming increasingly important. Especially in more complex settings — see the finance and marketing examples in the following sections — our new uncertainty-handling technology can exacerbate this problem of handling massive data, because MCDB's Monte Carlo computations can be highly CPU-intensive. For wide adoption of the MCDB approach to uncertainty, it is therefore very desirable to provide the MCDB functionality on an affordable, massively parallel platform.

- The MCDB prototype incorporates a major reworking of the standard relational query-processing engine. Because of the effort required, it is unlikely that this technology will be directly incorporated into commercial relational database products; an alternative path to market is needed.

- Since most real-world data is not stored in relational databases, it is important to be able to deal with data in a wide variety of formats.

To address the first issue, we note that Monte Carlo computations can be performed independently for each tuple bundle, and Monte Carlo replications for a given tuple bundle can be executed independently of each other. Thus the MCDB computations have the potential to be massively parallelized. MC^3 realizes this potential: the excellent scalability and ease of parallel programming made possible by the map-reduce approach are ideal for our purposes. Our MC^3 prototype uses Hadoop, an open-source implementation of Google's map-reduce processing framework. Hadoop's map-reduce has been shown to be highly scalable, as demonstrated by Yahoo's recent Daytona Terabyte sort record of 209 seconds using 910 servers.[1] Preliminary results from Google (68 seconds using 1000 servers) provide additional evidence.[2]

Our choice of Hadoop — as well as our choice of Javascript Object Notation (JSON) as the MC^3 internal data model — addresses the second issue raised above. The Hadoop infrastructure and JSON data format are becoming increasingly popular. Appealing features of Hadoop include fault tolerance and the capability to allocate and reallocate resources (CPU, memory, storage) as needed; this functionality allows massive parallelism to be achieved using commodity hardware, further increasing the appeal of the map-reduce approach. Although it appears hard to precisely delineate the class of queries that can be processed by MC^3, we believe that it is quite large: MCDB can handle virtually any BI SQL query, it appears that virtually any such query can be rewritten as a directed acyclic graph of operators (i.e., a query plan) that can be processed by MC^3. We speculate that many probabilistic XML queries can also be handled by MC^3. As a consequence of these considerations, techniques for managing uncertain data in this setting have the potential to be widely used.

With respect to the third issue, the use of JSON means that MC^3 can gracefully deal with data provided in non-relational formats, without any need to reformat the data prior to processing. Indeed, we obtain this functionality "for free" from the Hadoop platform. In this paper, we primarily deal with JSON data that can be viewed as reformatted probabilistic relational data. However, work on probabilistic XML data [19] leads us to believe that extensions to full-fledged probabilistic JSON data should be achievable. Such extensions would then permit MC^3 to interact with many recently developed repositories for scaled-out cluster environments, in which data attributes may be multi-valued and records in the same table may differ in their number of attributes [4, 7, 27, 29].

Extending the MCDB functionality to the map-reduce setting raises a number of challenging questions. How exactly do we map MCDB's tuple-bundle processing methods to Hadoop and JSON? Must we directly generate a query plan as a sequence of map-reduce operations, or can we facilitate this process via use of a higher level query language? What are the different ways in which MCDB queries can be parallelized, and for which scenarios are these various parallelization schemes effective? A key technical challenge is how to "seed" tuple bundles so as to generate streams of the pseudorandom numbers that form the basis of the Monte Carlo computations. For statistical correctness, the streams used by the various tuple bundles must be mutually disjoint. Seeding is challenging because it must be done in a highly parallel and distributed fashion, over an enormous number of tuple bundles, and without requiring storage of too much seeding information in each tuple bundle.

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

- We provide the first system for managing uncertain data in a map-reduce environment, showing how to represent MCDB tuple bundles as JSON arrays and how to translate an MCDB query plan to map-reduce.

- We show how query-plan generation can be facilitated by use of JAQL, an open-source language for querying JSON data.

- We identify two MCDB-specific parallelization schemes called inter-tuple and intra-tuple parallelism, show how to implement these schemes using map-reduce, and identify scenarios under which each scheme is effective.

- We develop and analyze an efficient distributed-seeding method called SeedSkip that is based on a random number generator with skip-ahead functionality, as well as a "fallback" method called SeedMult that is based on multiple pseudorandom number generators and can be used when SeedSkip does not apply.

- We show, via a set of experiments, that our parallelization techniques can yield linear scaleup when processing uncertain data, and that intra-tuple parallelism can provide linear speedup for certain very expensive VG functions.

[1] www.hpl.hp.com/hosted/sortbenchmark

[2] http://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html

We note that other parallel-processing platforms, data formats, and query languages can potentially be used to extend MCDB. Our goal was proof-of-concept, and our platform choices were partially made as a matter of convenience. We believe, however, that at least some of our techniques, and the lessons learned, are applicable to other possible extensions of the MCDB methodology to forward-looking information-management architectures.

Paper Organization. The remainder of the paper is organized as follows. Section 2 gives some background information. Section 3 gives an overview of how MCDB functionality is realized using map-reduce, JSON, and JAQL. Section 4 considers the distributed-seeding problem, and Section 5 presents our experimental study. We conclude the paper in Section 6.

2. BACKGROUND AND RELATED WORK

In this section, we review some basic facts about the MCDB system, the map-reduce parallel programming framework, and pseudorandom number generation.

2.1 The Monte Carlo Database System

MCDB [16], like most probabilistic-database prototypes, is based on *possible worlds* semantics for uncertain data. Each possible world corresponds to a possible concrete realization of all uncertain values in the database, and there is a probability distribution over the set of possible worlds, i.e., over the set of possible database instances.[3] Usually, this possible-worlds distribution is implicitly determined by explicit probability distributions that are associated with individual (sets of) uncertain attributes; in general, there can be statistical dependencies among attributes both within and between tuples. In this setting, a given SQL query over an uncertain database does not have a single, deterministic result, but rather there exists a probability distribution over the set of possible query results. This query-result distribution is jointly determined — usually in a very complicated fashion — by the possible-worlds distribution together with the query itself. The goal of querying over uncertain data is to compute or estimate interesting properties of the query-result distribution.

For example, consider a SQL query such as

```
SELECT SUM(AMOUNT) AS totSales FROM SALES
```

where `SALES(CID, AMOUNT)` is an uncertain table in which the `AMOUNT` attribute for each row, i.e., each customer, has a gamma distribution (values are rounded to the nearest dollar), with parameters that can depend on attributes such as customer ID or location. Then the result of this query is not a deterministic number that represents total sales, but rather a probability distribution over the possible values of total sales. We might be interested in features of this distribution such as its mean ("expected total sales"), standard deviation ("variability in total sales"), or 0.01-quantile ("approximate worst-case total sales"), or we may simply wish to plot the probability distribution. For set-valued queries, we might be interested in quantities such as the probability that a specified tuple appears in the query answer.

[3]When one or more of the uncertain attributes are real-valued, with uncertainty specified by a probability density function, the set of possible worlds can be uncountably infinite.

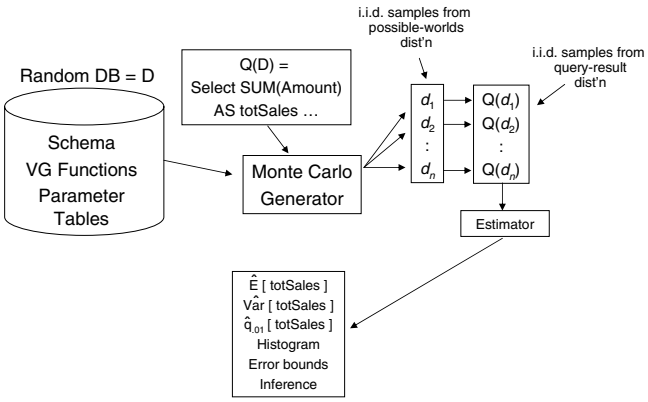

Figure 1: Query evaluation in MCDB.

MCDB does not try to compute characteristics of the query-result distribution exactly, but rather estimates these characteristics using a Monte Carlo approach. Figure 1 shows, conceptually, how MCDB evaluates a query. Each random table in an uncertain database is represented on disk by its schema, together with a set of VG functions that are used to generate realizations of uncertain attribute values, and an optional set of *parameter tables* (ordinary SQL tables) that are used to parameterize the VG function. Conceptually, when a query is issued, the MCDB system invokes the VG functions to generate a set of N (≥ 1) independent and identically distributed (i.i.d) samples from the possible-worlds distribution. We often paraphrase this process by saying that the system performs N *Monte Carlo replications* to produce N sampled possible worlds, which are also called *realizations* or *instantiations* of the database. The query is then evaluated over each of the sampled possible worlds, thereby generating a set of i.i.d. samples from the query-result distribution. These latter samples can be used to estimate characteristics of interest for this distribution.

E.g., consider the foregoing sum-of-sales scenario with 1000 Monte Carlo replications (i.e., with 1000 sampled possible worlds). Suppose that we have stored on disk a parameter table `CUST_ATTRS(CID, REGION)`, which lists the customers and the regions in which they live, as well as parameter tables `AMT_SCALE(REGION, SCALE)` and `AMT_SHAPE(CID, SHAPE)`, which store the scale and shape parameters for the gamma distributions. Observe that the scale parameter is the same for all customers within a given region, whereas the shape parameter varies between customers; thus there is an implicit statistical dependency between customers living in the same region. This simple gamma model might describe uncertainty associated with sales data currently stored in the database, or it might represent a predictive stochastic model that is extrapolating sales data into the future. The MCDB schema for the random table `SALES` essentially gives a recipe for assembling a sample realization of the table, specifying which VG function to invoke and how to glue the outputs of the VG function invocations together:

```
CREATE TABLE SALES(CID, AMOUNT) AS
 FOR EACH d in CUST_ATTRS
  WITH MONEY AS Gamma(
   (SELECT n.SHAPE FROM AMT_SHAPE n
    WHERE n.CID = d.CID),
   (SELECT sc.SCALE FROM AMT_SCALE sc
```

```
  WHERE sc.REGION = d.REGION))
SELECT d.CID, m.VALUE FROM MONEY m
```

We use the `Gamma` VG function from the MCDB library. This function takes as input two 1×1 tables that specify shape and scale parameters, and returns a 1×1 table (here named `MONEY`) that contains a pseudorandom sample from the corresponding gamma distribution. The `FOR EACH` clause instructs MCDB to loop over the list of customers stored in the `CUST_ATTRS` table. For each customer, a realization of the `AMOUNT` attribute is generated via a call to `Gamma`, which is first parameterized by executing the two SQL subqueries that extract the appropriate shape and scale parameters from the `AMT_SHAPE` and `AMT_SCALE` parameter tables. The final `SELECT` clause specifies how to assemble the row of the `SALES` table corresponding to the current customer from the outer loop.

Then (conceptually), MCDB will follow the above recipe to generate 1000 realizations of the `SALES` table and apply the SUM query to each realization to obtain 1000 values of total sales. The expected value and standard deviation of total sales can then be estimated as the average and sample standard deviation of these 1000 numbers. We can also use standard Monte Carlo techniques to characterize the precision of these estimates, e.g., by computing confidence intervals. Alternatively, we may simply want to create a histogram of the 1000 numbers in order to get an approximate picture of the total-sales density function.

The VG function given above simply generates samples from a standard probability distribution. MCDB offers a built-in library of such standard distributions, both continuous and discrete. For example, the `DiscreteChoice` function takes as input a parameter table containing a column of values and a column that gives a selection probability for each value; a call to `DiscreteChoice` then returns a value, randomly chosen according to its associated selection probability. (In this manner, MCDB, and by extension MC^3, can capture both "attribute-value uncertainty" and "tuple-inclusion uncertainty" as in systems like Trio, MayBMS, and Mystiq.) In general, user-defined VG functions can encapsulate highly complex stochastic models that represent extrapolated uncertain data that is not actually stored in the database. E.g., we give an example in Section 5 of a VG function that generates a sample from the probability distribution for the value of a European call option at its time of exercise. In this case, there is no closed form for this distribution, and samples are generated by executing a small dynamic simulation of the underlying stock. Even though a statistical expert may be needed to write the VG function, no special expertise is needed to run queries that use the VG function; in this manner, MCDB can encapsulate statistical complexity.

As mentioned previously, MCDB does not actually instantiate the database multiple times; the costs for such a naive approach would be exorbitant. Instead, MCDB executes a single query plan over a set of *tuple bundles*. Formally, a tuple bundle corresponding to N Monte Carlo replications is simply a set of N tuples having a common schema. A tuple bundle is usually represented as a vector in which each deterministic attribute appears once, and each uncertain attribute is represented by a nested vector of length N that records the value of the attribute in each of the Monte Carlo replications. The representation of a tuple bundle may contain additional attributes that facilitate query processing. In our running example, the tuple bundle t in table `SALES` corresponding to the customer with `CID` equal to 105 might have the form $t =$ `(105,(123.50,274.00,...,180.76), seed)`. Here the second component is a vector that contains the 1000 realized values of the `AMOUNT` attribute for customer 105. The final component `seed` is the pseudorandom number seed used by the VG function to generate the sale amounts. More generally, if customer 105 appeared in some, but not all, of the 1000 possible worlds, the tuple bundle would be augmented by a special random vector called `isPresent`. For $1 \leq i \leq 1000$, the ith component value of this vector is `true` if and only if customer 105 appears in the ith sampled possible world. Because pseudorandom number generation is a deterministic process when started from a fixed seed, the foregoing tuple bundle can often be stored in compressed form as $t =$ `(105, seed, isPresent)`. The `AMOUNT` values can be regenerated in a consistent manner whenever they are needed. Moreover, a predicate such as `CID < 105` can be applied to the compressed version of the tuple bundle; if such predicates filter out most tuple bundles and can be applied early in the query plan, then most of the Monte Carlo computations can be avoided. MCDB exploits these ideas to achieve acceptable processing overheads.

MCDB extends the classical database operations of select, project, join, duplicate removal, and aggregation to handle tuple bundles; these extensions typically involve manipulation of the `isPresent` vector. Several new MCDB operators that are pertinent to our discussion include the `Instantiate` and `Seed` operators. The `Instantiate` operator invokes the VG functions to transform a tuple bundle from compressed to expanded form. The `Seed` operator attaches one or more seeds to a tuple bundle, one per VG function. Each seed is unique to the (tuple bundle, VG function) pair, and is used by the VG function to generate a stream of pseudorandom numbers during instantiation.

The current MCDB prototype returns the answer to a query in the form of a table consisting of one or more "value" columns that jointly record all of the distinct result tuples produced during the Monte Carlo replications. For each such tuple, a `Fraction` column records the fraction of replications in which the tuple appeared at least once. For the sum-of-sales example, the result table is a two-column table where the first column contains all of the distinct total-sales values; for each such value, the `Fraction` attribute is the fraction of replications having that value for the query answer. E.g., the tuples in the table might be $(\$100, 0.2), (\$150, 0.5), (\$200, 0.3)$ if 20% of the replications resulted in a query answer of \$100, and so forth. Then the expected total sales can be computed as $\$100 \times 0.2 + \$150 \times 0.5 + \$200 \times 0.3 = \$155$. Other features of the query-result distribution, such as the variance or 0.9-quantile of total sales, can also be computed from the MCDB result table; the table values can also be plotted as a histogram. In general, the error of such estimates decreases (in an appropriate probabilistic sense) at rate $O(N^{-1/2})$, where N is the number of Monte Carlo replications. Depending on the specific type of estimation involved, however, slower or faster rates of decrease can be observed; see [11] for a broad discussion of these and other issues related to simulation efficiency.

2.2 Map-reduce

MC^3 evaluates queries using Hadoop's [12] map-reduce infrastructure, an open source implementation of Google's

parallel programming framework [8]. Using map-reduce, MC^3 queries can be evaluated using large clusters comprising thousands of commodity servers. Map-reduce was inspired by the *map* and *reduce* functions commonly found in functional programming languages. The programmer need only specify a *map* and *reduce* function, and the infrastructure then takes care of parallelization, fault-tolerance, resource allocation, and distributed synchronization.

A map-reduce job takes as input a collection of key-value records and produces a collection of output values. To specify the desired data processing, the user defines a *map* function and a *reduce* function, whose operation is described below. In the first step of map-reduce processing, the input records are partitioned among one or more mapper tasks. Each mapper applies *map* to the input records in its partition, one record at a time. The *map* function takes as input a key-value record (k, v) and produces a collection of intermediate key-value records, $[(k'_1, v'_1), \ldots, (k'_n, v'_n)]$. For each distinct intermediate key k, all intermediate records having this key are collected to form a group $(k, [v''_1, \ldots, v''_m])$, and the groups are partitioned among a set of reducer tasks. This processing is accomplished by taking each intermediate key-value record produced by a mapper and routing to a reducer (often over a network) according to the key value; the reducer then sorts incoming pairs by key value to form the groups. Each reducer applies *reduce* to the groups in its partition, one group at a time. The *reduce* function takes a group as input and returns a final value for the group, often using an aggregation operation on the values in the group. The set of values returned by the calls to *reduce* is written out and comprises the output of the map-reduce job. Both mapper and reducer tasks operate in parallel. Thus the map-reduce framework evaluates a job in a manner similar to the way a query is processed in a shared-nothing parallel DBMS [10].

As a simple example, consider the problem of counting the frequency of each distinct word in a document. Here an input record to a mapper is a line of text. (This is the "value" part of the input key-value pair; for this example no input key is needed or used.) The mapper task tokenizes the line of text into words, and creates an intermediate key-value pair for each word, where the key is the word itself, and the value is equal to 1. Thus, all pairs corresponding to a given distinct word, say "jump," are sent to the same reducer to create a group of the form ("jump", $[1, 1, \ldots, 1]$). The reducer task simply adds up the values (i.e., the 1's) to determine the word frequency.

To avoid network bottlenecks, map-reduce provides *combiners* that allow a mapper to perform aggregation operations that would otherwise be performed by a (possibly remote) reducer. The partially aggregated data is then fetched by the reducers, which complete the aggregation. In our example, a given mapper's combiner function might aggregate all of the mapper's intermediate key-value records that correspond to a given word, creating a new key-value record whose key is the word and whose value is the number of local copies of the word. These partially aggregated results would then be sent to reducers, which would then add up the values as before. Thus the desired word frequencies are produced, but fewer pairs need to be sent from mappers to reducers. This feature is similar to the partial aggregation functionality in a parallel relational DBMS.

2.3 Pseudorandom Number Generation

In general, a pseudorandom number generator (PRNG) 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 $s_{i+1} = T_1(s_i)$. At each step, the generator typically uses a second transformation to create a 32-bit integer $r_i = T_2(s_i) \in \{1, 2, \ldots, 2^{32} - 1\}$, 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_1 and T_2 depend on the specific generator, as do the number of bits in each s_i (typically a multiple of 32). For a good generator, the sequence $u_1, u_2, \ldots$ is statistically indistinguishable from a sequence of "true" i.i.d. samples from the uniform$[0, 1]$ distribution, in the sense that the sequence will pass statistical tests for uniformity and randomness [15, Ch. 3]. The uniform pseudorandom numbers can then be transformed into pseudorandom numbers having a user-specified distribution [9]. The sequence of seeds produced by a PRNG eventually loops back on itself — that is, $s_C = s_0$ for some integer C with $s_i \neq s_j$ for $0 \leq i < j < C$ — thus forming a *cycle* of seeds. The number C is called the *cycle length* of the PRNG. Typically, the cycle contains all possible seed values, although some generators have multiple cycles that are sometimes exploited for parallel pseudorandom number generation [23, 31].

When a VG function is invoked in order to instantiate a tuple bundle, the function is "fed" an initial seed s_i that was originally attached to the bundle by the `Seed` operator. The VG function then calls a PRNG multiple times to consume a sequence of k seeds $s_i, s_{i+1}, \ldots, s_{i+k}$ during the course of the instantiation operation. Depending on the VG function, the number k of seeds consumed might or might not be easily predictable in advance. It is necessary that the seed sequences for the tuple bundles be mutually disjoint, since overlaps cause unintended statistical dependencies between bundles, which can lead to incorrect query results. Since the number of tuple bundles can run to the billions, and each bundle can consume millions of seeds, the cycle length of the underlying PRNG must be very long. Fortunately, state-of-the-art PRNGs can achieve cycle lengths of 2^{500} and longer, while retaining good statistical properties; see Section 4. A key challenge is that these long cycle lengths are achieved by using very long seeds, which can greatly inflate the size of the tuple bundles. Such inflation can significantly slow down query processing by inducing large processing overheads when moving the data around.

As discussed in Section 4 below, MC^3 needs even more random number streams than MCDB when the intra-tuple parallelism scheme is used, and both seeding and subsequent stream generation must be done in a highly parallelizable manner. A substantial literature has developed around the problem of generating pseudorandom number streams in parallel [6, 23, 30, 31], largely motivated by scientific applications. Unfortunately, none of the previously proposed methods is directly applicable to our problem. For many prior algorithms, the number of streams needed coincides with the number of processors, and hence is on the order of 10^1 to 10^3; our setting can require on the order of 10^6 to 10^9 streams. "Leapfrog" methods — in which successive seeds from a base generator are dealt out to the streams like playing cards from a deck — are simple but can suffer from statistical anomalies [6]. A number of methods use a simple linear congruential generator as the base generator,

which yields an inadequate cycle length for our purpose. The most promising parallel PRNGs appear to be the SPRNG generator of Mascagni et al. [23], and the PLFG generator of Tan [31], each of which uses a lagged Fibonacci generator as the base generator and exploits special properties of this class of generators. A key drawback of such generators is that the seed corresponds to a "lag table" that typically comprises tens of thousands of 32 bit integers and will not fit into a tuple bundle. As indicated in the previous paragraph, we take the classical "cycle splitting" and "independent sequences" approaches using huge-period generators, and limit the amount of seeding information in a tuple bundle by employing novel techniques for combining generators and for skipping ahead on a cycle.

3. SYSTEM OVERVIEW

MC^3 uses JSON for its data model and JAQL for query specification. JSON is a simple self-describing data format for semi-structured data [18], and JAQL is a high-level query language designed for JSON data [17]. The key appeal of JAQL is its ability to automatically generate a query plan of map-reduce jobs from a high level query specification. MC^3 makes extensive use of JAQL's user-defined functions to specify VG functions, distributed seeding, and tuple bundle operations. Since JAQL is designed for JSON, it is natural for MC^3 to store input and output data in JSON format, as well as using JSON during query processing to represent MCDB tuple bundles. As indicated previously, other data models, storage formats, and query languages can potentially be used to implement MCDB in the map-reduce setting. We hope to study the various trade-offs in future work.

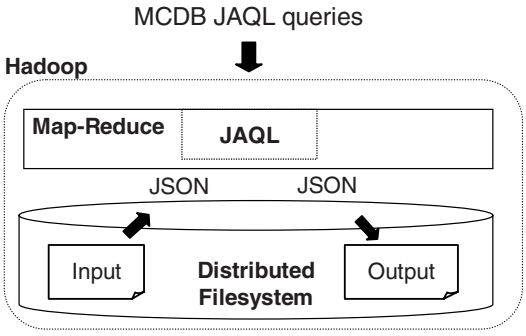

Figure 2: High-level architecture for MC^3.

The MC^3 architecture is illustrated in Figure 2. Data is assumed to be stored in files in the Hadoop Distributed File System (HDFS) and formatted using JAQL's binary JSON format. Note that data can be stored using other formats such as text-based JSON and XML. So long as the data can be converted on-the-fly to or from a JSON representation (see the bottom two arrows in the figure), it can be processed by JAQL. MCDB queries are specified using JAQL and automatically translated into a directed acyclic graph of map-reduce jobs. Hadoop's map-reduce is then used to evaluate query plans in parallel.

In the following sections, we show in detail how JSON is used during processing of the sum-of-sales query discussed in Section 2.1, how the query is specified using JAQL, and finally how the query is evaluated in parallel using map-reduce.

3.1 JSON

JSON was designed to be a simple, self-describing data format for representing commonly used programming language data structures. It consists of two complex types, arrays and records, and atomic types, i.e., number, string, boolean, and null. An array is a list of JSON values and is used to represent programming language arrays or lists. A record associates a set of field names with JSON values and is used to represent structs, dictionaries, or hash-tables.

Consider the sum-of-sales example from Section 2.1. The input data comprises the datasets `cust_attrs`, `amt_shape`, and `amt_scale`. This data is essentially relational, and we assume that each dataset is represented in JSON by an array (delimited by brackets) of records (delimited by braces). E.g., `cust_attrs` is represented as the array

$$\big[\{\texttt{cid: 102, region: NewEngland}\},$$
$$\{\texttt{cid: 226, region: Midwest}\},\ldots\big].$$

Figure 3 shows examples of in-process JSON data after the `Seed` and `Instantiate` stages. Prior to the `Seed` stage, the three input files have been joined to associate VG parameters (shape and scale) with each customer. The output of `Seed` is an array of JSON records in which a seed value has been added to each record. The `Instantiate` operator uses the seed value to initialize its PRNG and simulate possible worlds using a VG function. After `Instantiate`, each record corresponds to an expanded tuple bundle; e.g., in this example, the `amount` field has been instantiated.

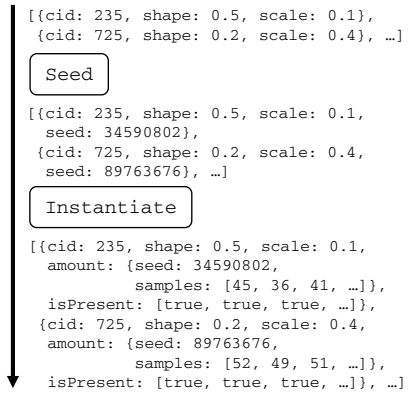

Figure 3: Example JSON data transformed by MC^3.

As mentioned previously, the current paper focuses on uncertain data that can be viewed as probabilistic relational data. This translates into a class of JSON data for which the uncertainty resides in the leaf data, e.g., in the atomic values. In general, uncertainty may exist in the structure, rather than just leaf data. For example, there may be uncertainty in the number of fields or types per record, resulting in complex probabilistic tree structures. A complete formal definition and set of processing methods for uncertain JSON data remains as future work.

That said, we note that, within the class of uncertain data that MC^3 can process — which subsumes the general MCDB

uncertainty model — MC^3 can easily handle a variety of non-relational input-data formats. As indicated above, MC^3 does not require data to be converted and stored prior to query execution, or to be loaded into a separate HDFS system for parallel processing. As long as a repository can partition data and provide an iterator that produces key-value pairs for each partition, the repository can be used by map-reduce, and consequently MC^3, to evaluate MCDB queries in parallel.

While some data organizations may lead to more efficient queries, the perspective taken by MC^3 is to favor flexibility in processing data. As a proof-of-concept, we reorganized TPC-H benchmark data to nest lineitems under orders. Then, we took Query Q4 from the original MCDB paper [16] and modified the JAQL query to process the nested data design. The key observation was that MC^3 was easily able to evaluate Q4 on nested data. However, performance was slightly worse when compared to the equivalent query that used the original, normalized TPC-H design. On the other hand, when nesting lineitems under partsupp, Q4 ran faster than with the normalized design.

3.2 JAQL

We illustrate the key features of JAQL using our running example. Specifically, the JAQL statements in Figure 4 specify the sum-of-sales query.

At line 1, the `read` operations for the inputs are assigned to variables. Starting at line 2, the records flow through a pipeline, are transformed by various operators, and are finally written to `result`. This part of JAQL's syntax has been influenced by UNIX pipes: instead of flowing bytes, a JAQL pipe flows JSON. The three input collections (i.e., JSON arrays) `cust_attrs`, `amt_shape`, and `amt_scale` are joined at line 2 using the `join` operator.

Seeding is implemented at line 3 by a `transform` operator that transforms each input customer record (as at the top of Figure 3) by invoking the user-defined `GetSeed` function and appending the result to the record as a `seed` field. The methods used to implement `GetSeed` are discussed in detail in Section 4. The `transform` operator simply describes the desired result of a transformation of a JSON input data object (such as a JSON array representing a tuple bundle); the `$` variable refers to the incoming JSON data object, allowing references to the object's components.

`Instantiate` is implemented at line 4 by invoking the `GenAmounts` VG function to simulate the possible worlds. This VG function is parameterized by scale and shape parameters associated with each customer, as well as the parameter `1000`, which specifies the number of samples (i.e., sale amounts) to produce, and the pseudo-random number seed. The output of this VG function — and the result of the JAQL `transform` operation — is simply a JSON array of 1000 samples, one for the customer's sale amount in each sampled possible world; call such an array a "sale array."

To compute the final answer, we must first sum the sale arrays over all customers to obtain an array of 1000 numbers that represents the total sum of sales in each of the 1000 sampled possible worlds; call this the "total-sales array." To this end, we employ the `ArraySum` function in line 5, which is a user-defined aggregate that sums multiple arrays and outputs a single array. The `group` operator invokes the `ArraySum` operator once on the entire input. (As discussed below, the `ArraySum` function is typically invoked within the context of the partial aggregation mechanism described in Section 2.2.)

At line 6, we have obtained the total-sales array, and we feed this array into the `Distribution` user-defined aggregation function. This function produces a JSON array of `(Value, Fraction)` pairs that represents for each unique total-sales value the fraction of sampled possible worlds in which the value appears. The array corresponds exactly to the MCDB result table discussed at the end of Section 2.1, and might have the form

$$\big[\{\texttt{Value: 100, Fraction: 0.20}\},\ \{\texttt{Value: 150, Fraction: 0.50}\},\ldots\big].$$

As discussed in Section 2.1, this empirical distribution can be used to estimate quantities such as the mean value, variance, or quantiles of the total-sales distribution. Finally, in line 7, the distribution is written to `result`.

```
1. $cust = read(hdfs('cust_attr'));
   $shape = read(hdfs('amt_shape'));
   $scale = read(hdfs('amt_scale'));
2. join $shape, $cust, $scale
   where $shape.cid == $cust.cid
     and $cust.region == $scale.region
   into { $shape, $scale }
   // Seed
3. → transform { $.*, seed: GetSeed() }
   // Instantiate: generate an array of 1000 samples
4. → transform GenAmounts($.seed, $.shape, $.scale, 1000)
   // Sum all sales tuple bundles
5. → group into ArraySum($)
   // Compute the distribution
6. → transform Distribution($)
7. → write(hdfs('result'));
```

Figure 4: JAQL for sum-of-sales example.

3.3 Using Map-Reduce for Parallelism

In this section, we discuss two parallelization schemes that are applicable to MCDB workloads: *inter-tuple* and *intra-tuple* parallelism. Inter-tuple parallelism is obtained by evaluating multiple tuple bundles in parallel; all of the possible worlds for a tuple bundle are generated by a single CPU core. Inter-tuple parallelism is naturally supported by map-reduce's partitioned parallelism. When there are fewer tuple bundles than cores, however, some cores will be idle. This problem becomes particularly acute when the Monte Carlo replications are computationally expensive — as in the European call-option example of Section 5 — or many replications are needed per tuple bundle. In such cases, use of intra-tuple parallelism allows Monte Carlo replicates for a single tuple bundle to be evaluated by multiple cores. We first illustrate inter-tuple parallelism via the sum-of-sales query, and then discuss intra-tuple parallelism.

3.3.1 Inter-Tuple Parallelism

The map-reduce plan for the sum-of-sales query is shown in Figure 5; recall that each *map* or *reduce* box in the plan may correspond to multiple actual mappers or reducers that are invoked in parallel during query processing. Job 1 joins `cust_attr` with `amt_shape` using `cid` as the join key. Similarly, Job 2 joins the output of Job 1 with the `amt_scale` using `region` as the join key. Joins are implemented with map-reduce by repartitioning the input files on the join key (a *map* operation) and joining records that have the same join key (a *reduce* operation). See [5] for a discussion of al-

ternative join techniques using map-reduce. Each mapper in Job 3 seeds its input tuple bundles, instantiates each tuple bundle by using the `GenAmounts` function, and invokes the `ArraySum` combiner interface (see Section 2.2) to compute one or more arrays. Each such array — which we call a "partial-sales array" — corresponds to aggregated sales values over a subset of the customers processed by the mapper, with one entry per sampled possible world. A single reducer for Job 3 fetches the partial-sales arrays from the Job 3 mappers and invokes `ArraySum` to aggregate the partial-sales arrays into the total-sales array. The reducer then computes the final empirical distribution (set of `Value-Fraction` pairs) and writes out the result.

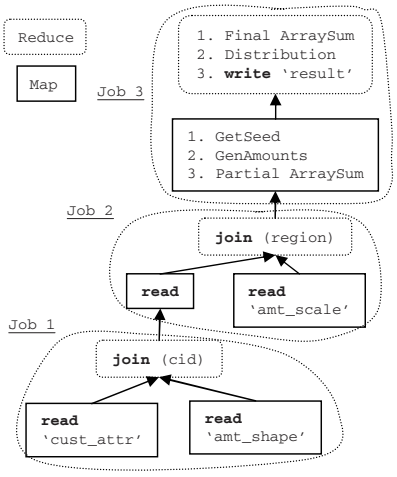

Figure 5: Map-reduce jobs for sum-of-sales example.

Just as in relational query optimization, there are usually multiple equivalent plans for a given JAQL query. E.g., in the plan of Figure 5, seeding and instantiation occur in the map stage of Job 3. However, instantiation can just as easily be scheduled to occur in the reduce stage. Currently, deciding which query plan to use is largely determined by the JAQL user; a focus of future work is to automate such tasks.

3.3.2 Intra-Tuple Parallelism

The map-reduce framework provides several possibilities for implementing intra-tuple parallelism. One approach is to use multiple threads per process; each thread is responsible for computing a *chunk* of sampled possible worlds for each tuple bundle. Using this approach, multiple processes are able to simulate a single tuple bundle's possible worlds. However, the degree of parallelism is limited by the number of CPU cores available at each server. An alternative approach, used by MC^3, is to slice each tuple bundle into multiple bundles, where each bundle represents the tuple's value in a chunk of sampled possible worlds. With a slight abuse of terminology, we also call these multiple tuple bundles "chunks." A chunk ID and seed are stored with each chunk. We then apply the inter-tuple parallelism processing method, but on the chunks rather than on the original tuple bundles. By using this latter approach, the degree of parallelism for a single tuple bundle can be as high as the total number of cores in the cluster, rather than the number at a single node.

The number of chunks per bundle depends on the available system resources. Outside of some minor modifications, seeding and instantiation use map-reduce in the same manner as with the inter-tuple parallelism approach. After the multiple chunks of a single tuple bundle are all instantiated, they are merged back into a single instantiated tuple bundle. If, for efficiency reasons, the deterministic attributes (e.g., `CID`) are not copied into each chunk during slicing, then these attributes must be merged as well.

4. DISTRIBUTED SEEDING

This section describes methods for seeding tuple bundles and for subsequently using the seeding information for purposes of tuple-bundle instantiation. We first give an algorithm, called SeedSkip, that is based on recent skip-ahead methods for huge-period PRNGs. SeedSkip guarantees that there will be no overlaps between streams of PRNG seeds and requires only a small amount of seeding information per tuple. Because the skip-ahead functionality is rather complex and has not yet been widely adopted, and because the method requires knowledge about VG-function properties that may not be readily available, we provide a fallback algorithm, called SeedMult, that uses multiple PRNGs and requires neither skip-ahead methods nor detailed knowledge about the VG functions. The SeedMult method does not completely rule out the possibility of overlaps between streams, but we show that the probability of such overlaps is vanishingly small.

For simplicity, we first assume that each tuple bundle needs to invoke only a single VG function during instantiation. We assume as in Section 3.3 that seeding corresponds to a mapper task in the map-reduce framework, and we assume that there are exactly J (≥ 1) logical processes, i.e., mappers, available to execute, in parallel, the seeding operation. We denote these mappers by $P_0, P_1, \ldots, P_{J-1}$. Also for simplicity, we initially assume that the inter-tuple parallel processing scheme is used for the instantiation step, so that a single mapper generates all of the Monte-Carlo realizations of a tuple bundle. Under the foregoing assumptions, each tuple bundle needs to be seeded with exactly one seed.

4.1 The SeedSkip Method

We first give a brief overview of skip-ahead techniques, and then illustrate how we can exploit these techniques for seeding.

4.1.1 Skip-Ahead Techniques for PRNGs

A PRNG with seed cycle $s_0, s_1, \ldots, s_{C-1}$ supports skipping-ahead if for one or more values of $\nu > 1$ it implements an efficient transformation $T^{(\nu)}$ such that $T^{(\nu)}(s_i) = s_{i+\nu}$ for all i. I.e., the generator can skip ν positions forward on the cycle without having to generate the intermediate seeds $s_{i+1}, s_{i+2}, \ldots, s_{i+\nu-1}$. Typically, one must predetermine a desired skip length ν and then precompute a set of generator-dependent "skip parameters" that support the desired skip length. This precomputation is usually expensive and can be quite complicated for modern huge-period generators but, once the parameters are computed, skips of length ν can be performed very efficiently, so that the precomputation cost is amortized over many skips.

For example, many popular types of PRNGs belong to the family of "combined multiple recursive generators" (CMRGs). An individual multiple recursive generator (MRG)

has a linear transformation function T. Specifically, for a seed s that is a vector of k 32-bit integers, an MRG recursively sets $s_n = As_{n-1} \bmod m$ for a $k \times k$ matrix A and large integer m. A CMRG maintains k MRGs in parallel, where typically $k \leq 4$, and produces the overall random integer r_n at the nth step by combining the k seeds $s_n^1, s_n^2, \ldots, s_n^k$, e.g., by setting $r_n = (s_n^1 + s_n^2 + \cdots + s_n^k) \bmod m'$ for an appropriate value of m'. L'Ecuyer [20] has developed an efficient technique for skipping ahead by a fixed number of steps ν in a CMRG. The idea is to precompute the matrix $A^\nu \bmod m$ using a "divide and conquer" algorithm, and then exploit the fact that, for an MRG, $s_{n+\nu} = (A^\nu s_n) \bmod m = (A^\nu \bmod m)(s_n) \bmod m$. As another example, consider the recent WELL512 generator of Panneton et al. [24]. WELL512 is our huge-period generator of choice, because it has a period of 2^{512} and improves on the statistical properties of the Mersenne Twister, a well known huge-period PRNG. WELL512 is also based on a linear recurrence, but with the modulus equal to 2, so that bits are manipulated individually. Efficient skip-ahead techniques for such generators have very recently been proposed in [13, 14]; these techniques are nontrivial to implement, involving the representation of the linear transformation $s \mapsto As$ as a formal series, appropriately manipulating the series coefficients to represent the skip, and then effecting the actual skips using the modified series. In this case, for a given skip length, one precomputes and stores the modified coefficients that correspond to the skip.

4.1.2 Seeding with SeedSkip

The SeedSkip method uses a single PRNG for both seeding and instantiation — denoted by G and chosen as the WELL512 generator in our implementation — and requires that we can upper bound by k^* the number of bundles seeded per mapper and, most importantly, by ρ^* the number of G seeds used up during any single invocation of any VG function. (Recall that each tuple bundle invokes the VG function $\omega = N$ times, where N is the number of Monte Carlo replications.) For example, in the call-option examples of Section 5, if we use a bounded-seed algorithm for generating normal variates, such as the polar method [9], then we know exactly how many seeds are required to generate a value for an option, and can set ρ^* equal to this number.

SeedSkip trivially "seeds" a tuple bundle with its mapper number and intra-mapper tuple-bundle identifier. Call these quantities n_p and n_i, and note that $n_p \in [0, J-1]$ and $n_i \in [0, k^* - 1]$. To naively instantiate a bundle, we would then generate $n_p k^* + n_i$ skips of length $\gamma = \omega\rho^*$, which will get us to the starting point on the G cycle for the instantiation. (To actually do the instantiation, we then use up at most γ seeds for the bundle.) In fact, we can vastly reduce the number of required skips by storing a carefully selected set of skip parameters, using $O\big(\log_2(C/\gamma)\big)$ storage, where C is the cycle length of G.

Specifically, suppose that $C/\gamma = 2^j$ for some integer j. Compute skip parameters that correspond to skip lengths $\nu = \gamma, 2\gamma, \ldots, 2^{j-1}\gamma$. Then $O\big(\log_2(n_p k^* + n_i)\big)$ skips along the G cycle are required for each tuple bundle to get to the starting point for the bundle's instantiation. I.e., denoting by $b_1 b_2 \cdots b_j$ the binary representation of $n_p k^* + n_i$, execute a skip of length $2^{b_1}\gamma$ if and only if $b_1 = 1$, then execute a skip of length $2^{b_2}\gamma$ if and only if $b_2 = 1$, and so forth.

4.2 The SeedMult Method

The SeedSkip method cannot be used if a good implementation of the advanced skip-ahead technology for huge-period generators is not readily available or, more importantly, if the VG functions are so complicated that it is impossible to obtain a reasonable upper bound on the number of seeds consumed during a VG function call. In this case, the SeedMult seeding method can be used instead. In the following, denote by k_j the number of tuple bundles that mapper P_j is assigned to seed, and by l_i the number (typically unknown to the user) of successive seeds consumed by tuple bundle t_i during instantiation.

The SeedMult method uses four PRNGs, denoted G^1–G^4; generators G^1 and G^2 are used for seeding, and generators G^3 and G^4 are used for instantiation; we assume that G^4 is a huge-period generator. We use generators having different internal structure, so that the pseudorandom number sequences generated by the different generators can be viewed as statistically independent for purposes of random number quality and analysis; this "independent sequences" approach is common in the literature; see, e.g., [23]. A seed s_j^i for generator G^i $(i = 1, 2, 3, 4)$ is an array $(s_{j,1}^i, s_{j,2}^i, \ldots, s_{j,m_i}^i)$ of m_i 32-bit integers. The jth execution of G^i updates the current seed s_{j-1}^i via a transformation $s_j^i = T_1^i(s_{j-1}^i)$ and, using a second transformation, creates a 32-bit integer $r_j^i = T_2^i(s_j^i) \in \{1, 2, \ldots, 2^{32} - 1\}$, which is normalized to a pseudorandom uniform number u_j on $[0, 1]$ via $u_j^i = r_j^i / 2^{32}$. PRNG G^i has cycle length N_i.

Each mapper P_j $(0 \leq j \leq J-1)$ executes the following procedure for seeding its local tuple bundles.

1. Initialize generator G^1 with a fixed seed s_0^1. (The same seed is used by all mappers.)
2. Generate $m_2(j+1)$ successive random integers $r_0^1, r_1^1, \ldots, r_{m_2(j+1)-1}^1$ from G^1, and use the last m_2 of these integers to form an initial seed $s_0^2(j)$ to use with G^2. I.e., $s_0^2(j) = \big(r_{jm_2}^1, r_{jm_2+1}^1, \ldots, r_{(j+1)m_2-1}^1\big)$.
3. Use G^2 to generate a sequence of $k_j m_3$ random integers $r_0^2(j), r_1^2(j), \ldots, r_{k_j m_3 - 1}^2(j)$, and use the ith subsequence of length m_3 to seed the ith local tuple bundle. More specifically, setting $K_0 = 0$ and $K_l = k_1 + k_2 + \cdots + k_l$ for $l \geq 1$, seed tuple bundle t_{K_j+i} with $s_0^3(K_j + i) = \big(r_{im_3}^2(j), r_{im_3+1}^2(j), \ldots, r_{(i+1)m_3-1}^2(j)\big)$ for $0 \leq j \leq J-1$ and $0 \leq i \leq k_j - 1$.

Later, at instantiation time, the following procedure is used to instantiate tuple bundle t_i, which has previously been seeded with seed $s_0^3(i)$ as computed above.

1. Initialize G^3 with seed $s_0^3(i)$, and use G^3 to generate m_4 successive random integers to form a seed $s_0^4(i)$.
2. Initialize G^4 with $s_0^4(i)$, and use G^4 to generate the l_i random numbers needed by the `Instantiate` operation for t_i.

4.2.1 Analysis

The following result is key to our analysis of the SeedMult algorithm.

LEMMA 1. *Consider a PRNG having a cycle of length C — comprising seeds $s_1, s_2, \ldots, s_C$ — and having seed transformation function T. Fix $K > 1$ and let $M_1, M_2, \ldots, M_K$*

be K mutually independent random variables, each uniformly distributed on $\{1, 2, \ldots, C\}$. For $k = 1, 2, \ldots, K$, define a segment σ_k of length L (≥ 1) by setting $\sigma_{k,1} = s_{M_k}$ and $\sigma_{k,l} = T(\sigma_{k,l-1})$ for $2 \leq l \leq L$. Then the probability that the K segments have one or more overlaps is less than $2K^2L/C$.

PROOF. For $2 \leq j \leq K$, denote by A_j the event that segments σ_1 through σ_j are mutually non-overlapping. Then the probability of interest can be written as $1 - P(A_K)$. Note that σ_k will overlap σ_j if s_{M_k} coincides with one of the $2L - 1$ seeds $s_{M_j-L+1}, s_{M_j-L+2}, \ldots, s_{M_j+L-1}$. (Here subscripts are to be interpreted modulo C.) Given that the first $j - 1$ segments are mutually non-overlapping, the probability that σ_j overlaps one or more of these segments is maximized when $\sigma_1, \sigma_2, \ldots, \sigma_{j-1}$ are arranged such that each pair of adjacent segments is separated by at least $L-1$ seeds. This arrangement maximizes the number of "bad" positions to which s_{M_j} can be assigned, namely $2L - 1$ bad positions for each of the $j-1$ prior segments. Thus, denoting by A^c the complement of event A, we have

$$P(A_j \mid A_{j-1}) = 1 - P(A_j^c \mid A_{j-1}) \geq 1 - \frac{(j-1)(2L-1)}{C},$$

so that, using the Bernoulli inequality,

$$\begin{aligned}
1 - P(A_K) &= 1 - P(A_2)P(A_3 \mid A_2)P(A_4 \mid A_3) \\
&\quad \cdots P(A_K \mid A_{K-1}) \\
&\leq 1 - \left(1 - \frac{2L-1}{C}\right)\left(1 - \frac{2(2L-1)}{C}\right) \\
&\quad \cdots \left(1 - \frac{(K-1)(2L-1)}{C}\right) \\
&\leq 1 - \left(1 - \frac{(K-1)(2L-1)}{C}\right)^{K-1} \\
&\leq 1 - \left(1 - \frac{(K-1)^2(2L-1)}{C}\right) \\
&= (K-1)^2(2L-1)/C,
\end{aligned}$$

and the desired result follows immediately. □

We can now analyze the probabilities of overlaps during the seeding and instantiation operations. For the seeding algorithm, observe that, by construction, the initial G^2-seeds $s_0^2(0), s_0^2(1), \ldots, s_0^2(J-1)$ for the J mappers are based on mutually disjoint segments of the G^1 cycle. These J seeds for G^2, can be viewed as being randomly placed, uniformly and independently, on the cycle for G^2. Each seed $s_0^2(j)$ initiates a segment of $k_j m_3$ successive seeds on the G^2 cycle. With $k^* = \max_{0 \leq j \leq J-1} k_j$, Lemma 1 implies that the probability of any overlaps between these segments is less than $\alpha_2 = 2J^2k^*m_3/C_2$. For the instantiation algorithm, set $K = k_0 + k_1 + \cdots + k_{J-1}$ and observe that the K seeds $s_0^3(0), s_0^3(1), \ldots, s_0^3(K-1)$ can be viewed as being randomly placed on the cycle for G^3, and the probability of any overlaps between segments (each having length m_4) is less than $\alpha_3 = 2K^2m_4/C_3$. Finally, let $l^* = \max_{1 \leq i \leq K} l_i$ be the maximum number of seeds consumed by a tuple bundle during instantiation, and view the K seeds $s_0^4(0), s_0^4(1), \ldots, s_0^4(K-1)$ as being randomly placed on the cycle for G^4. Since each seed initiates a segment of length at most l^*, the overlap probability is less than $\alpha_4 = 2K^2l^*/C_4$.

With suitable choice of generators, m_3 can be chosen to be a small integer, so that seeding a tuple bundle does not unduly increase the bundle size. Moreover, α_2, α_3, and α_4 can be made vanishingly small, so that, effectively with probability 1, no seed is ever used more than once for any of the generators. Specifically, we can use the generators in Table 1. When $J = 2^{10} \approx 1{,}000$, $k_i = 2^{20} \approx 1{,}000{,}000$ for $1 \leq i \leq J$ (so that there are $K = 2^{30}$, or about 1 billion tuple bundles in total), and $l_i = 2^{20}$ for $1 \leq i \leq K$, then, for the above choice of generators, we have $\alpha_2 \approx 2^{-148} \approx 10^{-44}$, $\alpha_3 \approx 2^{-56} \approx 10^{-17}$, and $\alpha_4 \approx 2^{-431} \approx 10^{-129}$. Clearly, the probability of any overlaps is negligible, and only four 32-bit integers of seeding information need be stored in each tuple bundle. The selected generators, besides having the requisite seed sizes and cycle lengths, are also known to have good statistical properties.

i	Came	Ref.	Seed length (m_i)	Cycle length (C_i)
1	LCG16807	[25]	1	$\approx 2^{31}$
2	MRG32k3a	[21]	6	$\approx 2^{191}$
3	CLCG4	[22]	4	$\approx 2^{121}$
4	WELL512a	[24]	16	$\approx 2^{512}$

Table 1: Generators for seeding and instantiation.

Scheme	Integers stored	PRNG calls: seeding	PRNG calls: instantiation setup
SeedMult	4	≈ 4	16
SeedSkip	2	0	≤ 30

Table 2: Per-bundle space and time costs for seeding and instantiation. "Setup" refers to the process of reaching the starting point on the G^4 cycle.

4.3 Performance Comparison of Methods

To get a better feel for the performance tradeoffs between the two seeding methods, suppose that we have $J = 1000$ mappers and an upper bound of $k^* = 1$ million bundles per mapper, and that we use a single VG function and $\omega = 10^4$ Monte Carlo replications. Also suppose that we use the generators as in Table 1. Then the space and time costs for the SeedMult and SeedSkip schemes are as in Table 2. Note that the seeding cost for the SeedMult scheme is dominated by the calls to G^2; the per-bundle amortized cost of G^1 calls is negligible. Also note that the results in the first two columns are independent of J and k^*.

Thus the SeedSkip scheme requires less seeding information per tuple bundle and has lower seeding costs. The instantiation setup can require more PRNG calls, but this may not be an actual disadvantage if calls to G^4 are faster than calls to G^3. Indeed, calls to WELL512 are exceedingly fast because the seed transformation function is implemented using bit operations such as XOR. In contrast, CLCG4 uses integer arithmetic, and is somewhat slower. The cost of computing skip parameters is unique to SeedSkip, but this cost can often be amortized over multiple queries; see Section 4.4 below. In our experiments (see Section 5), we found that SeedSkip was slightly faster than SeedMult, but the other map-reduce processing costs masked the performance differences between the seeding schemes. The foregoing performance considerations — together with the fact that SeedSkip guarantees no overlaps between seed sequences — indi-

cate that SeedSkip, when applicable, is the method of choice. This is why we view SeedMult as a fallback method.

4.4 Extensions

We now drop some of our simplifying assumptions, and show how the algorithms described above can be extended in several important ways.

Intra-Tuple Parallelism. As discussed in Section 3.3.2, the intra-tuple parallelism scheme uses multiple mappers to generate the N Monte Carlo realizations of a tuple bundle during instantiation. I.e., the set of Monte Carlo replications is divided into δ (> 1) chunks of $\omega = N/\delta$ replications each, and these chunks are generated in parallel (by a set of J' mappers). Inter-tuple parallelism corresponds to the special case $\delta = 1$. We modify the S-PPRNG and SeedMult algorithms to assign each chunk its own seed; our previous descriptions and analyses still hold, except that now each k_j is interpreted as the number of chunks that are seeded by mapper P_j, and ω is interpreted, as above, to be the number of Monte Carlo replications per chunk.

Multiple VG Functions. Suppose that M (> 1) VG functions must be seeded per tuple bundle. As with intra-tuple parallelism, our prior algorithms and analyses carry over virtually unchanged, except that now each (VG function, chunk) pair is assigned a unique seed. With a slight abuse of terminology, we henceforth use the term "chunk" to denote such a pair — thus there are $M\delta$ chunks in total for each tuple bundle — and k_j is again interpreted as the number of chunks that are seeded by P_j.

Shared Storage. If the J mappers have access to a (small) amount of shared storage, as is the case for Hadoop, then speedups for SeedMult are possible by amortizing seeding costs over multiple queries. For example, suppose that the number of seeding mappers used per query is bounded above by J^* over the query workload, and the upper bound k^* on the number of chunks per mapper also holds for the workload. Then the seeds $s_0^2(0), s_0^2(1), \ldots, s_0^2(J^*-1)$ can be computed once and written to shared storage. These seed values can simply be looked up at seeding time (in parallel, since only read operations are involved). Similarly, the SeedSkip scheme can also exploit shared storage, to amortize the initial cost of computing the skip-ahead information over a query workload. E.g., if the values of J and k^* in Section 4.3 serve as upper bounds for the query workload, then we can precompute and store the 30 sets of skip parameters for use by all of the queries.

Hybrid Algorithms. Hybrids of SeedMult and SeedSkip can be developed by using skip-ahead methods to place starting seeds on the G^2 and/or G^3 cycles in an equally-spaced manner, rather than dropping them randomly. We leave a detailed study of these methods for future work.

5. EXPERIMENTAL EVALUATION

We evaluated the performance of MC^3 based on *scaleup* and *speedup* metrics, which are commonly used to evaluate parallel database systems [10]. Scaleup is determined by increasing the workload and system resources proportionally. Intuitively, k times as much hardware should be able to handle k times the workload. Speedup is determined by keeping the workload constant and increasing the system resources. Intuitively, k times as much hardware should be able to process the same workload in a fraction $1/k$ of the time.

First, we considered inter-tuple parallelism and used several queries from the original MCDB paper [16] to determine scaleup performance. MC^3 scaled up well in these experiments. Next, we considered intra-tuple parallelism for workloads that have small input files but can be CPU-intensive; specifically, we executed queries that involved uncertain stock-option values. The results demonstrated effective speedup as CPU cores were added, since the workload was truly CPU-intensive, i.e., the VG functions were expensive.

Hardware and Software. All experiments were run on a cluster of ten servers. Each server had two 64-bit 2.1 GHz AMD quad-core CPUs, 16GB of memory, and eight 250 GB disks. The software used was 64-bit Linux V2.6.23.1-42, Sun Java JDK V1.7, Hadoop V0.18.1, and JAQL V0.3. Although such a cluster is small by map-reduce standards — due to constraints on our experimental setup — we expect our scaleup results to hold for much larger clusters, provided that the appropriate parallelization scheme is used.

5.1 Inter-tuple Parallelism

To evaluate scaleup, we measured the running times of several queries on increasingly large data sets while proportionally increasing the number of servers. In the ideal case, we would expect to see the running time remain the same as the data set and number of servers are proportionally scaled up.

5.1.1 Experimental Setup

(a) Data and Query. In our experiments, we used the TPC-H benchmark dataset.[4] The dataset was generated using the *dbgen* program provided by the benchmark, converted to JSON, and loaded into Hadoop's HDFS. We used queries Q1 and Q4 from the original MCDB paper [16] which are described for convenience below:

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. Once 1996 is generated, the additional revenue is computed.

The query plan consisted of seven map-reduce jobs. Three jobs were used to obtain products supplied by Japanese companies by joining `NATION` with `SUPPLIER`, `PARTSUPP`, and `LINEITEM`. Two more jobs were used to seed the records in `ORDERS`, join them with customer orders, and instantiate them to produce hypothetical orders. The two remaining jobs were used to join Japanese-supplied products with the hypothetical orders and to compute the empirical distribution.

Query Q4. This query is the "marketing query" mentioned in Section 1, which estimates the effect of a 5% customer price increase on an organization's profits. A complicated Bayesian VG function that involves acceptance-rejection sampling is used to predict a customer's demand at a new price; see [16] for details. An interesting feature of this query is that the Bayesian approach determines per-customer demand distributions, rather than the traditional one-size-fits-all approach of fitting a single demand curve for all customers. One can therefore easily estimate the effect

[4] http://www.tpc.org/tpch/

of a price change on various customer segments that are defined dynamically at query time via, e.g., SQL group-by or selection operations.

The query plan consisted of five map-reduce jobs. Two jobs were used to obtain the cost of supplied parts by joining `LINEITEM`, `ORDERS`, and `PARTSUPP`. The third job grouped `LINEITEM` by part to parameterize parts with their prior distribution. The fourth job seeded supplied costs, joined supplied costs with prior parameters, and instantiated the tuple bundles. The final job computed the probability distribution.

(b) Experimental Procedure. We varied the number of servers from 1 to 10 and prepared 10 corresponding datasets, from 2GB for 1 server to 20GB for 10 servers. For each computing-cluster size, we evaluated both $Q1$ and $Q4$ and recorded total elapsed time. Each query was evaluated using both the SeedMult and SeedSkip distributed seeding methods, and 1000 Monte Carlo replicates were produced.

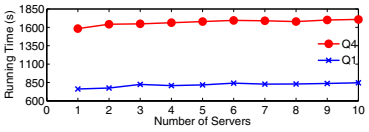

Figure 6: Scaleup results for Q1 and Q4

5.1.2 *Discussion*

Figure 6 displays the results for $Q1$ and $Q4$ using SeedSkip. The displayed execution times are the average of five runs. For both queries, the time for seeding was insignificant compared to the time for I/O, sorting, network transfer, and instantiation. As a result, we did not observe significant differences in overall elapsed times when SeedSkip or SeedMult were used. (In isolation, it appeared that SeedSkip was at least 5% faster than SeedMult.) For both $Q1$ and $Q4$, join operations accounted for nearly 50% of the total execution time. $Q1$ spent most of the remaining time on tuple-bundle manipulation (e.g., aggregating arrays) rather than on instantiation. In contrast, $Q4$'s VG function was more complex, so substantially more time was spent on instantiation than on tuple-bundle manipulation.

The network was not the bottleneck for either query. Both queries spent a large fraction of their time reading, writing, sorting, and transferring data. $Q4$ also spent substantial time on its CPU-intensive VG function. The curves in Figure 6 are relatively flat, which indicates that MC^3 scaled up well. Even though the data set used was small, we expect that MC^3 will effectively scaleup for larger data sets as well, based on map-reduce's excellent scalability properties.

5.2 Intra-tuple Parallelism

To evaluate the speedup of intra-tuple parallelism, we kept the workload constant and measured query execution time while increasing the number of servers. In the ideal case, we would expect to see the overall running time decrease by a factor of $1/k$ (so that the speedup factor equals k) when k times as many servers are used.

5.2.1 *Experimental Setup*

(a) Data and Query. In these experiments, we used a customer table and an option table. The option table contained four different call options and the customer table contained 100 customers. Each customer had 0 or more options. Both the option table and customer table were generated randomly. The query estimates the future portfolio value for each customer after the options are exercised. The VG function generates samples of option values by simulating the underlying stock values over the time period of interest. Two queries were executed, one for Asian call options and one for European call options. The Asian-option query uses a relatively inexpensive VG function, whereas the European-option query uses a very expensive VG function, about 20 times as expensive as the Asian-option VG function; see the Appendix for details.

(b) Experimental Procedure. For each query, we varied the number of cores from 4 to 80 and observed the execution times. Each query was executed using SeedMult seeding and 1000 Monte Carlo replication. Since four options were used, our baseline was to use four cores. At this setting, intra-tuple and inter-tuple parallelism are equivalent.

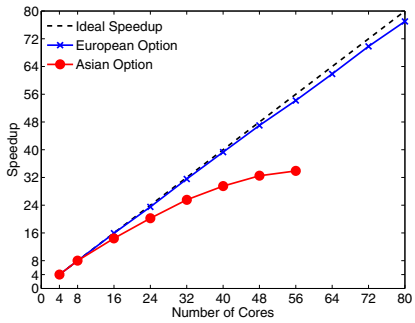

Figure 7: Speedup results for option queries.

5.2.2 *Discussion*

Figure 7 displays the speedup behavior for the two queries. The execution times for both queries were dominated by CPU time. Since the data set was so small, I/O, network, and sorting consumed a much smaller fraction of time relative to the inter-tuple parallelism experiments.

Ideally, we expect a straight line for speedup. We notice that for the expensive European-option query, the speedup is nearly ideal. Such excellent results can be attributed to the fact that a large fraction of the execution time was due to VG function evaluation, so that intra-tuple parallelism, which directly tries to reduce this execution cost, is very effective.

In contrast, the speedup curve for the Asian-option query starts to flatten out after about 20 cores. The reason for this degradation is that the task of merging chunks has a fixed cost, independent of the number of cores, as opposed to the VG-function execution costs, which decrease as the number of cores increases. Thus, as the number of cores increases and the VG function execution costs decrease, the merging cost eventually dominates the total execution cost, thereby limiting the benefits of parallelism. I.e., the Asian-option query was too inexpensive for intra-query parallelism to be helpful at higher degrees of parallelism.

These experiments indicate the importance of matching the parallelization scheme to the query and data. If the correct scheme is chosen, good scaleup can be obtained.

6. CONCLUSIONS AND FUTURE WORK

The MC^3 system subsumes MCDB functionality, and extends it to a map-reduce setting, opening up the potential for massive parallelization and flexible data handling when managing uncertain enterprise data. Our experiments show that, using our proposed parallelization schemes and novel SeedSkip and SeedMult distributed seeding methods, MC^3 indeed can scale up effectively to take advantage of available computing resources and efficiently process data even with complex extrapolation-based uncertainty. Our experiments show, however, that one must be careful in matching the parallelization scheme to the task at hand. Inter-tuple parallelism is most appropriate for large numbers of tuples and relatively cheap VG functions, whereas intra-tuple parallelism becomes advantageous as the number of tuples decreases or the VG-function cost increases. Thus "classical" probabilistic-database applications, where uncertainty is caused by data integration, information extraction, sensor errors, and so forth, will most likely be well suited to inter-tuple parallelism, whereas applications that involve complex, expensive Monte Carlo operations may be better suited to intra-tuple parallelism.

One notable feature of the map-reduce approach is the relative ease and speed at which we were able to get MCDB queries up and running; based on our experience, we expect that the Hadoop environment will be well suited to trying out new ideas and prototypes for uncertain data management.

There are many possible directions for future research. Some issues surrounding Monte Carlo approaches to managing uncertain data are common to both MCDB and MC^3. These include estimation of extreme quantiles, allowing the user to pre-specify the precision of Monte Carlo estimates, and methods for handling inter-table correlations; see [16] for a discussion. An issue specific to MC^3 is the lack of techniques for optimization of MC^3-oriented JAQL queries, e.g., with respect to the choice of parallelization scheme. Current JAQL optimization techniques are entirely based on query rewrites, and are both ad hoc and not tailored to MC^3. In general, the processing efficiency of MCDB, JAQL, and MC^3 queries can vary, and is not yet well understood. This topic presents rich opportunities for future work.

Another interesting topic is the development of formal semantics and practical processing methods for general uncertain JSON data; our current techniques are limited to nested JSON data for which only leaf values are uncertain. Finally, an intriguing feature of Hadoop is the potential for implementing dynamic simulation techniques, e.g., methods for simulating possible worlds until a desired precision is obtained, or for simultaneously simulating competing business policies and redirecting map-reduce resources away from a policy as soon as it becomes apparent that the policy is inferior.

Overall, the MC^3 system lies at the confluence of multiple interesting new technologies. Our examples have emphasized uncertainty arising from stochastic models in the hope that this work will encourage new information-management applications in which traditional analytical methods are extended seamlessly to data-intensive settings, thereby improving overall analytical accuracy and decisionmaking.

Acknowledgements

The authors wish to thank Rainer Gemulla and the referees, whose comments improved the paper. Preparation of this paper was supported at least in part by the National Science Foundation under Grant No. 0803511.

APPENDIX

Details of the Call-Option Queries

To save space, we describe the examples using MCDB terminology throughout. Consider a set of customers, each owning a few different types of *call options*. All customers in the database have bought the options at the same time. A buyer of option 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's profit is 0. Thus the value of the option O_i at time T is $\big(P_i(T) - K_i\big)^+$, where $x^+ = \max(x, 0)$.

We first consider Asian call options with Black-Scholes dynamics. Let $S_i(t)$ denote the value of the ith underlying stock at time t and 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 (so that different stocks evolve independently), r_i is the mean rate of return, and σ_i is the stock's "volatility." The payoff for the Asian option is the average price over a sequence of $m \geq 1$ equally spaced measurement times in $[0, T]$: $P_i(T) = m^{-1}\sum_{j=1}^{m} S_i(t_j)$, 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 $S(t_j) = S_i(t_{j-1}) \times \exp\Big((r_i - 0.5\sigma_i^2)(t_j - t_{j-1}) + \sigma_i\sqrt{(t_j - t_{j-1})}Z_{ij}\Big)$ 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.

The database has parameter tables `CUST(CID,OID,NUM)`, which specifies the number of options of each type owned by each customer, and `OPTION(OID,INITVAL,R,SIGMA,K,M,T)`, which specifies the properties of each option. We define the following random table of option values at future time T:

```
CREATE TABLE OPTION_VAL(OID,VAL) AS
 FOR EACH o in OPTION
  WITH OVAL AS VAL_COMP (
   VALUES(o.INITVAL, o.R, o.SIGMA, o.K, o.M, o.T))
  SELECT o.OID, v.VALUE FROM OVAL v
```

The VG function `VAL_COMP` simulates the stock price for an option O_i over $[0, T]$, computes the payoff $P_i(T)$, and then computes the option value at time T. The following query computes the value of the option account for each customer.

```
SELECT c.OID, SUM(c.NUM * ov.VAL)
FROM CUST c, OPTION_VAL ov
WHERE c.OID = ov.OID GROUP BY c.OID
```

We next consider European call options with time-varying volatility. The setup is almost the same as above, but now the payoff is simply the terminal stock price, i.e., $P_i(T) = S_i(T)$, and the stock price now evolves according to the modified equation $dS_i(t)/S_i(t) = r_i\,dt + a_i\big(S_i(t)\big)^{1/2}\,dW_i(t)$ for

some constant $a_i \in [0,1]$. Thus the volatility varies over time, depending on the stock price. This equation does not have an analytic solution in general, and must be simulated. The standard Euler method approximates the dynamics by dividing up the interval $[0,T]$ into m small steps of length $\Delta t = T/m$ and computing the dynamics according to the recursive equation $S_i(t_j) = S_i(t_{j-1}) + r_i S(t_{j-1})\Delta t + a_i\big(S_i(t_{j-1})\big)^{3/2}\sqrt{\Delta t}Z_{ij}$, where $t_j = j\Delta t$ and each Z_{ij} is a normal sample as before. Thus we can formulate an MCDB query similar to the previous example, by appropriately modifying the VG function and adding the a_i attribute to the `OPTION` parameter table. For both examples, observe that the cost of the VG function increases as the parameter m increases. In our experiments, we choose a large value of m, so that the VG function is expensive.

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

[3] J. Boulos, N. N. Dalvi, B. Mandhani, S. Mathur, C. Ré, and D. Suciu. MYSTIQ: a system for finding more answers by using probabilities. In *ACM SIGMOD*, pages 891–893, 2005.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI*, pages 15–15, 2006.

[5] H. chih Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD*, pages 1029–1040, 2007.

[6] P. D. Coddington. Random number generators for parallel computers. *The NHSE Review*, 2, 1996.

[7] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB*, pages 1277–1288, 2008.

[8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

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

[10] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

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

[12] Hadoop. http://hadoop.apache.org/core/.

[13] H. Haramoto, M. Matsumoto, and P. L'Ecuyer. A fast jump ahead algorithm for linear recurrences in a polynomial space. In *SETA*, pages 290–298, 2008.

[14] H. Haramoto, M. Matsumoto, T. Nishimura, F. Panneton, and P. L'Ecuyer. Efficient jump ahead for $\mathbb{F}_2$-linear random number generators. *INFORMS J. Computing*, 20(3):385–390, 2008.

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

[16] R. Jampani, F. Xu, M. Wu, L. L. Perez, C. M. Jermaine, and P. J. Haas. MCDB: a Monte Carlo approach to managing uncertain data. In *ACM SIGMOD*, pages 687–700, 2008.

[17] JAQL. http://code.google.com/p/jaql/.

[18] JSON. http://www.json.org.

[19] B. Kimelfeld and Y. Sagiv. Modeling and querying probabilistic XML data. *SIGMOD Record*, 37(4):69–77, 2008.

[20] P. L'Ecuyer. Random numbers for simulation. *Comm. ACM*, 33(10):85–97, 1990.

[21] P. L'Ecuyer. Good parameters and implementations for combined multiple recursive random number generators. *Oper. Res.*, 47(1):159–164, 1999.

[22] P. L'Ecuyer and T. H. Andres. A random number generator based on the combination of four LCGs. *Math. Comput. Simul.*, 44(1):99–107, 1997.

[23] M. Mascagni. Some methods of parallel pseudorandom number generation. In R. Schreiber, M. Heath, and A. Ranade, editors, *Algorithms for Parallel Processing*, pages 277–288. Springer, 1997.

[24] F. Panneton, P. L'Ecuyer, and M. Matsumoto. Improved long-period generators based on linear recurrences modulo 2. *ACM Trans. Math. Software*, 32(1):1–16, 2006.

[25] S. K. Park and K. W. Miller. Random number generators: Good ones are hard to find. *Comm. ACM*, 31(10):1192–1201, 1988.

[26] C. Re and D. Suciu. Managing probabilistic data with MystiQ: The can-do, the could-do, and the can't-do. In *SUM*, pages 5–18, 2008.

[27] SimpleDB. http://aws.amazon.com.

[28] S. Singh, C. Mayfield, S. Mittal, S. Prabhakar, S. Hambrusch, and R. Shah. Orion 2.0: native support for uncertain data. In *ACM SIGMOD*, pages 1239–1242, 2006.

[29] SQLServer Data Services. http://www.microsoft.com/sql/dataservices/default.mspx.

[30] A. Srinivasan, D. M. Ceperley, and M. Mascagni. Random number generators for parallel applications. In *Monte Carlo Methods in Chemical Physics*, pages 13–36. Wiley, 1997.

[31] C. J. K. Tan. The PLFG parallel pseudo-random number generator. *Future Generation Computer Systems*, 18:693–698, 2002.

[32] D. Z. Wang, E. Michelakis, M. N. Garofalakis, and J. M. Hellerstein. BayesStore: managing large, uncertain data repositories with probabilistic graphical models. *Proc. VLDB*, pages 340–351, 2008.