MCDB and SimSQL: Scalable Stochastic Analytics within the Database

Peter J. Haas
IBM Research–Almaden
phaas@us.ibm.com

Chris Jermaine
Rice University
cmj4@rice.edu

ABSTRACT

This short paper highlights published work on the MCDB database system as well as its successor, SimSQL. These database systems are designed from the ground up to support *stochastic analytics*; that is, data analysis performed with the aid of stochastic simulations.

1. INTRODUCTION

Dealing with data uncertainty has always been a challenge, and never more so than in the presence of the "four V's" of Big Data: volume, variety, veracity, and velocity. Data uncertainty comes from many sources: measurement errors, because data have not yet been observed and must be forecast (think of sales figures for the upcoming quarter), or because they are missing and must be imputed. The first step in addressing this uncertainty is to quantify it by means of a statistical model; the model can then be queried to answer questions of interest. Importantly, the answer to a query over uncertain data—e.g., "What would my profits have been last year had I raised my prices by 10%?" or "how many days until all orders placed today are delivered?"—has the form of a probability distribution over possible answers; inferring characteristics of this distribution (moments, quantiles, histogram approximations, and so on) is the primary goal. In the real world, there are as many statistical models as there are types of uncertainty. Thus the key challenge in dealing with big, uncertain data is building systems that can deal with a broad spectrum of models and can query these models when the amount of data is massive.

This paper highlights a seven-year joint effort between researchers at Rice University and IBM Research focused on developing precisely these types of systems. The philosophy is to use Monte Carlo techniques that can handle uncertainty in great generality and to exploit parallel processing technologies to handle massive datasets. The Monte Carlo technology is built into the fabric of the system, so that stochastic analysis occurs in the database, close to the data, avoiding the need to extract the data and process it in some external analytics tool. Our current system, SimSQL, is available at `http://cmj4.web.rice.edu/SimSQL/SimSQL.html`.

2. MCDB

2.1 Overview

Historically, the first system to be created was the *Monte Carlo Database System (MCDB)*, which handles many types of uncertain data, primarily within a traditional parallel-database framework [9, 10, 11], although the promise of MapReduce for Monte Carlo processing was recognized fairly early on [14]. MCDB allows an analyst to attach arbitrary stochastic models to a database, thereby specifying, in addition to the ordinary relations in the database, "random" relations that contain uncertain data. These stochastic models are implemented as user- and system-defined libraries of external C++ programs called *Variable Generation functions*, or VG functions for short. A call to a VG function generates a pseudorandom sample from a probability distribution; VG functions are usually parameterized on the current state of the non-random relations (tables of historical sales data in our example). Generating a sample of each uncertain data value creates a *database instance*, i.e., a realization of an ordinary database.

2.2 Example

For an example of how a VG function can be used to create a random relation, imagine that we have a database of hospital patients:

```
patients(name, gender)
```

We wish to simulate a systolic blood pressure for each patient. To do this, imagine that we have a parameter table that tells us, for each possible gender, the average systolic blood pressure, as well as the standard deviation:

```
sbp(mean, std, gender)
```

We can then create a random relation having a simulated blood pressure for each patient as follows:

```
create table sbp_data(name, gender, sbp) as
  for each p in patients
  with res as Normal (
    select s.mean, s.std
    from sbp s where s.gender = p.gender)
  select p.name, p.gender, r.value
  from res r
```

Briefly, what this code does is to consider every tuple `p` in the `patients` table. For each patient `p`, the `sbp` table is queried to find the mean and standard deviation associated with the patient's gender. The resulting tuple is then used to parameterize the `Normal` VG function, which returns a random tuple having one normally distributed attribute, `value`. The final `select` statement then composes that random tuple with the patient's `name` and `gender` to produce an output tuple. The tuples produced by all of the executions of the `select` statement (one for each patient) are `union`ed together to create the `sbp_data` relation.

This example is quite simple. In general, a VG function can be parameterized by taking multiple relations as input, where each input relation is computed by means of a subquery involving one or more of the non-random relations. Furthermore, each VG function can output an arbitrary number of correlated tuples, and a random `create table` statement can have multiple VG function invocations. The VG function presented in the example simply generates a sample from a standard distribution. More generally, samples

can be generated from arbitrarily complex stochastic models, so that underlying probability distribution need not have a closed-form representation. For example, a VG function might produce a random sample corresponding to the future value of a stock by internally running a stochastic financial simulation.

2.3 Queries Over Random Relations

Users of MCDB can design SQL queries that reference both random and non-random database relations. In the example of Section 2.2, one simple query of interest might be

```
select average(sbp) from sbp_data
```

Because `sbp_data` is a random relation, the "query result" is actually a probability distribution over possible query results, i.e., over possible values of the average systolic blood pressure. Our goal is to estimate features of interest for this probability distribution, such as moments, quantiles, and so on. Executing such a query in MCDB is equivalent to first generating a random database instance and then running the query over the instance. This generates a sample from the query-result distribution. Iterating this process N times generates N samples from this distribution, which can then be used to estimate specified distribution features such as the expected value or variance of the average blood pressure. We can even compute a histogram approximation of the entire query-result distribution. Replacing `average` by `max` in the the above query, we can estimate, e.g., the variance of the highest simulated systolic blood pressure in the database.

To ensure acceptable practical performance, MCDB employs 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 Monte Carlo iterations. Subsequent work [1, 11] has demonstrated how MCDB can be extended to deal with risk analysis (by efficiently estimating extreme quantiles) and to handle *threshold queries*, e.g., "Which regions will see more than a 2% decline in sales with at least 50% probability?".

3. SIMSQL

3.1 Overview

SimSQL [5] is a re-implementation and extension of MCDB. Unlike MCDB, SimSQL allows random database relations to be used when parameterizing VG functions. This allows SimSQL to implement hierarchical stochastic models such as latent Dirichlet allocation (LDA) [3] for learning topics from text.

Moreover, SimSQL allows both versioning and recursive definitions of random relations. For example, data in random relation `A` can be used to parametrize the stochastic generation of relation `B`, which in turn can be used to parametrize the stochastic generation of a second version of relation `A`, and so on. Whereas MCDB merely allowed generation of sample realizations of a given stochastic database D—i.e., realizations of a static database-valued random variable—the foregoing extensions enable SimSQL to generate realizations of a database-valued Markov chain $D[0], D[1], D[2], \ldots$. That is, the stochastic mechanism that generates a realization of the ith database state $D[i]$ may explicitly depend on the prior state $D[i-1]$.

3.2 Example

For an example of how SimSQL can be used to simulate a Markov chain, imagine that we have a directed graph, stored in the following relation:

```
edge(vert_from, vert_to)
```

We also have a number of people:

```
person(name)
```

We want to simulate a Markov chain that initially associates each person with a random vertex in the graph, and then has all of those people perform a random walk on the graph.

To code this stochastic model in SimSQL, we first define the random relation that holds everyone's initial position:

```
create table pos[0] (name, vert) as
  for each p in person
  with res as Categorical (
    select distinct vert_from, 1.0
    from edge)
  select p.name, r.value
  from res r
```

Here, the `Categorical` VG function chooses an item randomly from a set, with the probability of choosing an item proportional to a given weight. (Each vertex has weight `1.0` in our example.)

We then move those people around as follows:

```
create table pos[i] (name, vert) as
  for each p1 in person
  with res as categorical (
    select e.vert_to, 1.0
    from edge as e, pos[i-1] as p2
    where p2.name = p1.name and
      p2.vert = e.vert_from)
  select p.name, r.value
  from res r
```

This definition is similar to that of `pos[0]`, except that the next location for each person is chosen randomly from the set of locations reachable from the person's current location, where the current location is stored in `pos[i-1]`.

Similarly to MCDB, it is possible to pose queries over this sort of Markov chain. For example, we can ask "How many people are at each vertex at the 50th time tick of the chain?":

```
select count(*), vert
from pos[50]
group by vert
```

As with MCDB, SimSQL first generates a realization of the chain by generating `pos[0],pos[1],...,pos[50]` and then executes the `count(*)` query over this realization to produce a sample from the query-result distribution. This process is repeated N times and the N sample query results are used to estimate features of the distribution such as the expected number of people at each node at the 50th time tick.

As indicated by the above example, one potential application of SimSQL is to massive stochastic agent-based simulations. Indeed, Wang et al. [13] have shown that deterministic agent-based simulations can be viewed as a sequence of self joins for a relation in which each tuple contains the data for an agent. SimSQL allows this idea to be directly extended to a stochastic setting.

3.3 Application to Machine Learning

In this section, we focus on another key application of SimSQL: Bayesian Machine Learning (ML); see, e.g., [2].

Statistical ML In statistical ML, we first postulate a generative statistical model for a data set X, which implies a probability distribution function $f(X|\theta)$. Here θ represents the unseen model parameters and hidden variables; the elements of θ might correspond, for example, to the probabilities of different topics in a document or to the likelihood that a source of credit card transactions is fraudulent. *Learning* is the process of computing an estimate $\hat{\theta}$ of θ, based on the data X. In the Bayesian approach to learning, θ is viewed as

a random variable, and the user expresses his or her prior belief about θ by supplying a *prior distribution* $f(\theta)$. The goal of learning is to estimate features of $f(\theta|X)$, the *posterior distribution* of θ, given the data. Typical features of interest are the mode, i.e., the most likely value, and the mean. The posterior relates to the prior via *Bayes' Rule*: $f(\theta|X) \propto f(\theta)f(X|\theta)$, where $f(X|\theta)$ is the *likelihood* of the data, given θ. For industrial strength ML models, the posterior is high-dimensional and complex, and it is usually impossible to compute the posterior mode or mean analytically. A popular solution to this problem is to take samples from the posterior distribution and use these samples to estimate the feature of interest. For example, the average of the samples can be used as an estimate of the posterior mean. The most common sampling technique is *Gibbs sampling*.

Gibbs Sampling Basics Gibbs sampling [6] is a standard technique for generating samples from a high-dimensional probability distribution function $f(Y)$ that is known only up to a normalizing constant. In our setting, $Y = \theta$ and f is the posterior distribution of a complex Bayesian ML model. Gibbs sampling requires the simulation of a Markov chain, and so it is natural to code and execute Gibbs samplers using SimSQL.

A Gibbs sampler simulates a Markov chain whose stationary distribution is the desired target distribution. Briefly, the procedure works as follows. Assume that $Y = (Y_1, Y_2, ..., Y_d)$ is a $d-$dimensional vector of random variables. A key requirement of the Gibbs sampler is that, as is often the case, we can efficiently generate samples from the conditional distributions $f(Y_j|Y_{-j})$ for $j \in \{1..d\}$, where Y_{-j} is the vector comprising all variables except Y_j. To generate k samples the Gibbs sampler proceeds as follows:

1. Select an initial value $Y^{(0)} = (Y_1^{(0)}, Y_2^{(0)}, \ldots, Y_d^{(0)})$.
2. For $i = 1, 2, \ldots, k$, sample $Y_j^{(i)}$ $(j = 1, 2, \ldots, d)$ from $f(Y_j^{(i)}|Y_1^{(i)}, \ldots, Y_{j-1}^{(i)}, Y_{j+1}^{(i-1)}, \ldots, Y_d^{(i-1)})$.

After running the simulation for k steps of a "burn in" period, the current state of the chain can be taken as a sample from the target distribution.

Learning a GMM Using SimSQL For an example of how one can use SimSQL to execute a Gibbs sampler, consider the problem of learning a Bayesian Gaussian Mixture Model (GMM). A GMM views a data set as being produced by a set of K Gaussian (multi-dimensional normal) distributions; the kth Gaussian is parameterized by a mean vector μ_k and a covariance matrix Σ_k, and has an associated probability π_k. To produce $\mathbf{x}_j$, the jth point in the data set, the model first selects a Gaussian by generating a sampled vector $\mathbf{c}_j$ from a Multinomial$(\pi, 1)$ distribution—$c_{j,k}$ equals 1 if and only if the kth Gaussian is selected and equals 0 otherwise. The data point $\mathbf{x}_j$ itself is then sampled from the Gaussian indicated by $\mathbf{c}_j$. We put a Dirichlet(α) prior on π, a Normal(μ_0, Λ_0^{-1}) prior on each μ_k, and an InvWishart(v, Ψ) prior on each Σ_k. The learning goal is then to estimate all of the unseen c_j values, as well as all of the Gaussian parameters.

Denote by $\mathbf{p}_j^{(i)}$ the unit-length vector whose kth entry is proportional to $\pi_k^{(i)} \times \text{Normal}\left(\mathbf{x}_j|\mu_k^{(i)}, \Sigma_k^{(i)}\right)$ and by n the number of data points. A Markov chain to learn the desired posterior distribution can be derived as

$$\mu_k^{(i)} \sim \text{Normal}\left(\left(\Lambda_0\mu_0 + n\left(\Sigma_k^{(i-1)}\right)^{-1}\right)^{-1} \times \left(\Lambda_0\mu_0 + \left(\Sigma_k^{(i-1)}\right)^{-1}\sum_j c_{j,k}^{(i-1)}\mathbf{x}_j\right),\right.$$
$$\left.\left(\Lambda_0\mu_0 + n\left(\Sigma_k^{(i-1)}\right)^{-1}\right)^{-1}\right)$$

$$\Sigma_k^{(i)} \sim \text{InvWish}\left(n + v, \Psi + \sum_j c_{j,k}^{(i-1)}(\mathbf{x}_j - \mu_k^{(i)})(\mathbf{x}_j - \mu_k^{(i)})^T\right)$$

$$\pi_k^{(i)} \sim \text{Dirichlet}\left(\alpha + \sum_j c_j^{(i-1)}\right) \quad c_j^{(i)} \sim \text{Multinom}\left(\mathbf{p}_j^{(i)}, 1\right)$$

The task is to write a distributed code that simulates this chain.

The simulation can be implemented in SimSQL using a database schema with four random relations that correspond to the four classes of variables listed above:

```
clus_means[i](clus_id,dim_id, dim_value)
clus_covas[i](clus_id, dim_id1, dim_id2,dim_value)
clus_prob[i](clus_id, prob)
membership[i](data_id, clus_id)
```

The data to be processed are stored in a relation:

```
data(data_id, dim_id, data_val)
```

as are the various entries in the α vector (the "hyperparameter" for the Dirichlet prior on π):

```
cluster(dim_id, alpha)
```

Aside from this, the entire SimSQL code consists of (1) initialization codes for the first three random relations, (2) recursive definitions for all four random relations, and (3) a C++ implementation of the `multinomial_membership` VG function, which is used to update `membership[i]` (the other VG functions are all library functions). As an example, consider the following initialization:

```
create table clus_prob[0] (clus_id, prob) as
  with diri_res as Dirichlet
    (select clus_id, pi_prior
     from cluster)
  select diri_res.out_id, diri_res.prob
  from diri_res;
```

This code uses the hyperparameters stored in the `cluster` table to parameterize the Dirichlet VG function, which then outputs the value of $\pi^{(0)}$ as a set of $(k, \pi_k^{(0)})$ pairs, that are then stored in the `clus_prob[0]` table.

Here is an example of a recursive definition:

```
create table clus_prob[i](clus_id, prob) as
  with diri_res as Dirichlet
    (select cmem.clus_id,
       cmem.count_num+clus.pi_prior as diri_para
     from (select cm.clus_id as clus_id,
                  count(cm.data_id) as count_num
           from membership[i-1] as cm
           group by clus_id) as cmem, clus
     where cmem.clus_id = clus.clus_id)
  select diri_res.out_id, diri_res.prob
  from diri_res;
```

This code parameterizes the Dirichlet distribution by performing the required $\alpha + \sum_j c_j^{(i-1)}$ computation, in order to re-sample the selection probability for each of the clusters. This computation requires that we compute the number of times that each data point is assigned to each cluster, which is done via SQL aggregation.

3.4 SimSQL Implementation

SimSQL is a fully functional system, written mostly in Java. SQL codes are compiled by the system into a series of Hadoop MapReduce jobs. SimSQL supports most of the declarative part of SQL, including deeply nested, correlated subqueries. The SimSQL compilation process includes a full, cost-based logical optimization

(SimSQL's rule-based optimizer is written in Prolog) as well as physical optimization. The physical optimization procedure utilizes information such as table sizes, "interesting" sort orders, functional dependencies, and primary and foreign keys when choosing physical implementations of the relational operations. For example, SimSQL has five different distributed join implementations that it can choose from: a nested-loops join, a full sort-merge join, sort-one-side-only join, a merge-only join, and a pipelined, non-blocking join when one relation can fit into memory. SimSQL has been used on multi-terabyte-sized data sets distributed over 100 machines, and experiments in [4] indicate that the performance of SimSQL is competitive with, and in some cases superior to, other platforms for very large scale machine learning.

4. FUTURE WORK

SimSQL is an active research project, and work is ongoing. One important direction for future work is to explore the link between SimSQL and the general area of *probabilistic programming* [8], in which an end user specifies a generative stochastic model, e.g., a GMM, in a high-level programming language such as Church [7]. The code mixes standard programming language constructs (conditionals, loops, and so on) with statistical routines such as function calls that generate random values from a given distribution. The system then analyzes the code and automatically generates an executable inference algorithm—such as a Gibbs sampler—that infers the value of the unseen model parameters from observed data. The attraction of probabilistic programming is that it removes the need for a PhD-level computer scientist or statistician to derive and implement an inference algorithm; the derivation and implementation of the algorithm is done automatically, by a compiler.

Because SimSQL uses database techniques to take as input a very high-level, SQL-based specification of an inference algorithm, it makes a lot of sense to use SimSQL as a back end for a probabilistic programming framework. Like all database systems, SimSQL's SQL is declarative and "algorithms free", making it very attractive as a target for automatic code generation. In theory, software that automatically generates an SQL-based Markov Chain specification need not be concerned with efficiency, since the SimSQL optimizer and runtime will figure out how best to implement the chain.

Along these lines, we are currently working on a declarative, BUGS-like language [12] called MCL for specifying Markov chains, as well as a compiler that can translate such a specification into SimSQL's SQL dialect. This means that a user of SimSQL need not know SQL—the user need only understand a very simple, math-like language in order to code a Markov chain simulation. The next step will be to develop a compiler that can take as input a generative stochastic model, coded in MCL, and then automatically generate a Markov Chain specification (also in MCL) that can perform inference for the original stochastic model. This MCL specification can then be compiled, optimized, and executed over a very large data set by SimSQL. At this point, SimSQL will be a complete probabilistic programming platform for massive data.

We are also working on a number of systems-oriented problems in the context of SimSQL. For example, we are adding native support for vectors and matrices to the SimSQL engine. Currently SimSQL must, for example, represent a $d \times d$ matrix as d^2 different tuples, each a `(row, col, value)` triple. Encoding one million 100×100 matrices in this way results in 10 billion tuples, and moving all of these tuples around can be quite expensive. With native support for `vector` and `matrix` column types, SimSQL could store those one million matrices in one million tuples. We expect that this enhancement will speed up certain computations by as much as two orders of magnitude. We also plan to add special support to the SimSQL optimizer and execution engine to handle the new `vector` and `matrix` types. For example, the optimizer will be modified so that it understands the semantics of operations over vectors and matrices, and can automatically choose the best implementation for common computational patterns that appear in machine learning, for example, the construction of a Gram matrix from a table of vectors.

With the recent open source release of SimSQL, we expect that user experiences with the system will point the way toward new research and system challenges, as well as new applications. Our hope is to demonstrate the utility of designing a database system in which Monte-Carlo-based stochastic analytics are fundamental.

Acknowledgments In addition to the authors of this summary, many people have worked on MCDB and SimSQL, including Subi Arumugam, Zhuhua Cai, Jacob Gao, Ravi Jampani, Shangyu Luo, Luis Perez, Zografoula Vagena, Mingxi Wu, and Fei Xu.

5. REFERENCES

[1] S. Arumugam, R. Jampani, L. Perez, F. Xu, C. Jermaine, and P. J. Haas. MCDB-R: Risk analysis in the database. In *VLDB*, pages 782–793, 2010.

[2] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[3] D. M. Blei, A. N. Ng, and M. I. Jordan. Latent Dirichlet Allocation. *JMLR*, 3:993–1022, 2003.

[4] Z. Cai, Z. Gao, S. Luo, L. Perez, Z. Vagena, and C. Jermaine. A comparison of platforms for implementing and running very large scale machine learning algorithms. In *SIGMOD*, 2014. To appear.

[5] Z. Cai, Z. Vagena, L. L. Perez, S. Arumugam, P. J. Haas, and C. M. Jermaine. Simulation of database-valued Markov chains using SimSQL. In *SIGMOD*, pages 637–648, 2013.

[6] G. Casella and E. I. George. Explaining the Gibbs sampler. *The American Statistician*, 46(3):167–174, 1992.

[7] N. Goodman, V. Mansinghka, D. Roy, K. Bonawitz, and D. Tarlow. Church: a language for generative models. *arXiv preprint arXiv:1206.3255*, 2012.

[8] N. D. Goodman. The principles and practice of probabilistic programming. In *ACM SIGPLAN Notices*, volume 48, pages 399–402. ACM, 2013.

[9] R. Jampani, F. Xu, M. Wu, L. L. Perez, C. Jermaine, and P. J. Haas. The Monte Carlo Database System: Stochastic analysis close to the data. *TODS*, 36(3):1–41, 2011.

[10] 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 *SIGMOD*, pages 687–700, 2008.

[11] L. L. Perez, S. Arumugam, and C. M. Jermaine. Evaluation of probabilistic threshold queries in MCDB. In *SIGMOD*, pages 687–698, 2010.

[12] D. J. Spiegelhalter, A. Thomas, N. G. Best, W. Gilks, and D. Lunn. BUGS: Bayesian inference using Gibbs sampling. *Version 0.5,(version ii) http://www. mrc-bsu. cam. ac. uk/bugs*, 19, 1996.

[13] G. Wang, M. Vaz Salles, B. Sowell, X. Wang, T. Cao, A. Demers, J. Gehrke, and W. White. Behavioral simulations in MapReduce. *Proc. VLDB*, 3(1):952–963, 2010.

[14] F. Xu, K. S. Beyer, V. Ercegovac, P. J. Haas, and E. J. Shekita. E = MC^3: managing uncertain enterprise data in a cluster-computing environment. In *SIGMOD*, pages 441–454, 2009.