Simulation of Database-Valued Markov Chains Using SimSQL

Zhuhua Cai
Rice University
Houston, TX, 77251
caizhua@gmail.com

Zografoula Vagena*
LogicBlox, Inc.
Atlanta, GA, 30309
foula@acm.org

Luis Perez
Rice University
Houston, TX, 77251
lp6@rice.edu

Subramanian Arumugam
Rice University
Houston, TX, 77251
propus@gmail.com

Peter J. Haas
IBM Almaden
San Jose, CA, 95120
phaas@us.ibm.com

Christopher Jermaine
Rice University
Houston, TX, 77251
cmj4@rice.edu

ABSTRACT

This paper describes the SimSQL system, which allows for SQL-based specification, simulation, and querying of database-valued Markov chains, i.e., chains whose value at any time step comprises the contents of an entire database. SimSQL extends the earlier Monte Carlo database system (MCDB), which permitted Monte Carlo simulation of static database-valued random variables. Like MCDB, SimSQL uses user-specified "VG functions" to generate the simulated data values that are the building blocks of a simulated database. The enhanced functionality of SimSQL is enabled by the ability to parametrize VG functions using stochastic tables, so that one stochastic database can be used to parametrize the generation of another stochastic database, which can parametrize another, and so on. Other key extensions include the ability to explicitly define recursive versions of a stochastic table and the ability to execute the simulation in a MapReduce environment. We focus on applying SimSQL to Bayesian machine learning.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

Keywords

Markov chains; databases; machine learning

1. INTRODUCTION

SimSQL is a system that allows for SQL-based specification and large-scale, distributed simulation of database-valued Markov chains, i.e., chains whose value at any time step comprises the contents of an entire database. SimSQL employs many of the ideas first proposed in the context of the Monte Carlo database system (MCDB) [18], which is a prototype, SQL-based database system designed to support stochastic analytics. Both SimSQL and MCDB allow users to define—in addition to ordinary database tables—so-called *stochastic* database tables. In a stochastic table, some entries are random variables with associated probability distributions (not necessarily known in closed form) and others are ordinary, deterministic data values. Conceptually, when an SQL query is issued that includes stochastic tables, SimSQL and MCDB both use pseudo-random number generation techniques to instantiate each stochastic data value. Queries can be run over the resulting database instance (also called "possible world"); this process is repeated many times in a Monte Carlo fashion to obtain an empirical distribution of the query results. Both SimSQL and MCDB instantiate and process many possible worlds at the same time with high efficiency by sharing common costs using a "tuple bundle" approach.

SimSQL contains the basic functionality present in the original MCDB system, but also has some significant extensions. This paper focuses most closely on two key capabilities unique to SimSQL that are not present in MCDB and which significantly increase SimSQL's applicability vis-a-vis the earlier system.

1. SimSQL allows data in stochastic database tables to be used to parametrize the processes that stochastically generate the data in other stochastic database tables. This allows SimSQL to implement hierarchical stochastic models such as latent Dirichlet allocation (LDA) [5] for learning topics from text.

2. SimSQL allows both versioning and recursive definitions of stochastic database tables. For example, data in stochastic table `A` can be used to parametrize the stochastic generation of table `B`, which in turn can be used to parametrize the stochastic generation of a second version of table `A`, and so on.

Whereas MCDB merely allows generation of sample realizations of a given stochastic database D—in other words, 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 $D[i-1]$. Possible applications of this powerful functionality include Bayesian Machine Learning (ML) and massive stochastic agent-based simulations. In this paper we focus on ML applications.

*Work performed while Zografoula Vagena was affiliated with Rice University.

Material in this paper was supported by the National Science Foundation under grant number 0915315, and the Department of Energy under grant number DE-SC0001779.

SIGMOD'13, June 22–27, 2013, New York, New York, USA.

Scalable Bayesian Machine Learning in SimSQL. This paper gives several examples of how SimSQL can support Bayesian inference over large data sets. In Bayesian inference, it is assumed that a parametrized stochastic process generated the data. The parameters are uncertain, and the goal is to combine prior beliefs about the parameters with the observed data to infer distributional properties of the parameters. Depending upon the application, these parameters may describe clusters in the data, regression coefficients, or latent topics in text. Often, the "posterior distribution" on the parameters can be very difficult to describe analytically, so the most common analysis approach is to sample from the posterior distribution using Markov chain Monte Carlo (MCMC) simulation.

SimSQL's database-oriented approach gives a surprisingly appropriate set of abstractions for Bayesian ML. In most ML problems, there are a few classes of variables or parameters over which inference is to be performed. These might be latent variables (such as the identity of a cluster that produced a particular data point) or model parameters (such as the covariance matrices in a Gaussian mixture model). It is quite natural to group the members of such a class together and store them in a database table.

MCMC inference is natural in SimSQL because of the *VG function* abstraction that it borrows from MCDB. A VG function is a (nearly) arbitrary, (possibly) user-defined pseudo-random function that generates samples of uncertain data values; these samples are then used to construct sample realizations of stochastic tables. The sampling mechanism of a VG function depends on parameters that are provided as input tables. It is very easy to embed the sampling of the posterior distribution of a variable or parameter inside of a VG function, which is exactly what is needed for many MCMC algorithms. Furthermore, SQL is a very useful language for preparing complex parametrization of VG functions. This results in MCMC codes that are small and easy to understand "in the large"—that is, the specification for how variables fit together and parametrize each other is often remarkably simple.

Perhaps the most important advantage of running MCMC in SimSQL is that one can write SQL without worrying about how the code will be parallelized across many machines. SimSQL automatically takes care of the optimization and parallelization. SimSQL takes as input a specification written entirely in SimSQL's dialect of SQL, then optimizes it and executes the actual simulation in parallel on a Hadoop cluster, allowing for scalable processing.

Paper Highlights.

- This paper describes how SimSQL's dialect of SQL makes it very easy to declaratively specify Markov chain simulations.
- SQL codes for two Bayesian inference problems are given.
- Details are given on how SimSQL compiles Markovian simulations into computational plans that resemble classical database execution plans.
- Experiments show that SimSQL has reasonable performance for running large-scale, Markov chain simulations.

2. BAYESIAN LINEAR REGRESSION

To demonstrate, in a simple setting, the use of SimSQL for very large-scale Bayesian analysis, we describe how SimSQL can be used to implement MCMC inference for a very simple Bayesian linear regression model over a large database.

2.1 The Model

Suppose we have a large database table `myTable(x,y)` of n rows. Our goal is to learn two values a and b such that for an arbitrary observation $\mathtt{x}$, we can predict $\mathtt{y}$ via the formula $\mathtt{y} \approx a\mathtt{x} + b$. Although this is clearly a toy problem—in practice, with such a simple learning problem one is unlikely to resort to Bayesian methods—it makes a good first example.

To apply Bayesian methodology to such a problem, an analyst will first posit a stochastic generative process for the data:

1. $a \sim \text{Normal}(0, \sigma_0^2)$.
2. $b \sim \text{Normal}(0, \sigma_0^2)$.
3. $\sigma^2 \sim \text{InvGamma}(1, 1)$.
4. For each i in $\{1...n\}$, $\mathtt{y}_i \sim \text{Normal}(a\mathtt{x}_i + b, \sigma^2)$.

Here "$\sim$" should be read as "is sampled from" and InvGamma denotes the inverse gamma distribution.

We thus assume that the observed value $\mathtt{y}_i$ is normally distributed—with variance σ^2—around the predicted value $a\mathtt{x}_i + b$. The unknown parameters a and b are themselves treated as random variables and have normal "priors" with common variance σ_0^2 (which we take to be a known, constant "hyperparameter"). The term "prior" in Bayesian parlance refers to the probability distribution that we initially assume has produced the parameter values. The priors are based on previous knowledge or belief, or perhaps chosen for computational convenience. Our beliefs about the prior distribution are updated upon observation of the data in `myTable`, leading to a posterior distribution on the parameters; this posterior distribution is the ultimate object of our study. Choosing an appropriate generative process and priors is something of a "black art" and is beyond the scope of the paper. A reader who wants to learn more should look at an introductory text such as [4].

The specification of a stochastic generative process results in a probability density function (PDF) for the random data and parameters. In our example, we have:

$$\begin{aligned} &P(a, b, \sigma^2, \{\mathtt{y}_i\}|\{\mathtt{x}_i\}) \\ &= \text{Normal}(a|0, \sigma_0^2) \times \text{Normal}(b|0, \sigma_0^2) \\ &\quad \times \text{InvGamma}(\sigma^2|1, 1) \times \prod_i \text{Normal}(\mathtt{y}_i|a\mathtt{x}_i + b, \sigma^2) \end{aligned} \tag{1}$$

2.2 Inference

Our goal is to estimate the parameters a, b, and σ^2 after seeing the data in `myTable(x,y)`. That is, we wish to analyze the posterior distribution $P\big(a, b, \sigma^2|\{\mathtt{x}_i, \mathtt{y}_i\}\big)$. In practice, the posterior distribution is seldom amenable to closed-form analysis, and a common method for studying the posterior is to obtain samples from it that can be analyzed statistically. We accomplish this sampling using a particular MCMC algorithm called a "Gibbs sampler" [31]. Gibbs sampling is a technique for generating samples from a high-dimensional probability distribution function $P(X)$ that is known only up to a normalizing constant. A Gibbs sampler simulates a Markov chain whose stationary distribution is the desired target distribution. Briefly, the procedure works as follows: Assume that $X = (X_1, X_2, ..., X_d)$ is a $d-$dimensional vector of continuous and/or discrete random variables. A key requirement of the Gibbs sampler is that we can efficiently generate samples from the conditional distributions $P(X_j|X_{-j})$ for $j \in \{1..d\}$, where X_{-j} is the vector comprising all variables except X_j. To generate k samples the Gibbs sampler proceeds as follows:

1. Select an initial value $X^{(0)} = (X_1^{(0)}, X_2^{(0)}, \ldots, X_d^{(0)})$.
2. For each sample $i = \{1 \ldots k\}$, sample each variable $X_j^{(i)}$ from $P(X_j^{(i)}|X_1^{(i)}, \ldots, X_{j-1}^{(i)}, X_{j+1}^{(i-1)}, \ldots, X_d^{(i-1)})$.

```
CREATE TABLE tableA[0] (aValue) AS
  WITH   sampledVal AS Normal (SELECT * FROM VALUES (0, 10))
  SELECT *
  FROM   sampledVal;
CREATE TABLE tableB[0] (bValue) AS
  WITH   sampledVal AS Normal (SELECT * FROM VALUES (0, 10))
  SELECT *
  FROM   sampledVal;
```

Figure 1: Initializing a and b (σ_0^2 = 10).

```
CREATE TABLE variance[i] (s2Value) AS
  WITH sampledVal AS InvGamma (
    (SELECT 1 + dataNum/2
     FROM   metaData),
    (SELECT 1 + 0.5 *
            SUM ((y - aValue * x - bValue) *
                 (y - aValue * x - bValue))
     FROM   myData, tableA[i], tableB[i]))
  SELECT sampledVal.value
  FROM   sampledVal;
```

Figure 2: Updating the variance σ^2.

After running the simulation for k steps of a "burn in" period, the current state of the chain—which in our case would include $a^{(k)}$, $b^{(k)}$, and $\sigma^{2(k)}$—can be taken as a sample from the target distribution.

In in our example, we can use Bayes' rule together with the PDF in (1) to derive the following sampling distributions:

$$P(a|.) \propto \text{Normal}(a|0, \sigma_0^2) \times e^{(2\sigma^2)^{-1} \sum_i \{2(y_i - b)x_i a - x_i^2 a^2\}}$$

$$P(b|.) = \text{Normal}\Big(b \;\Big|\; \frac{\sum_i (\mathrm{y}_i - a\mathrm{x}_i)}{(\sigma^2/\sigma_0^2) + n}, \big(\frac{1}{\sigma_0^2} + \frac{n}{\sigma^2}\big)^{-1}\Big)$$

$$P(\sigma^2|.) = \text{InvGamma}\Big(\sigma^2 \;\Big|\; 1 + \frac{n}{2}, 1 + \frac{\sum_i (\mathrm{y}_i - a\mathrm{x}_i - b)^2}{2}\Big)$$

2.3 Learning the Model

A Gibbs sampler for this model must repeatedly update the values for a, b and σ^2 in a recursive fashion. The order in which to update these three parameters is arbitrary; we choose to first initialize a and b, and then update σ^2, a and b in sequence. We now describe how this recursion can easily be specified in SimSQL's SQL dialect.

Initialization. We begin by initializing the parameters a and b and storing each in its own one-row table: `tableA(aValue)` and `tableB(bValue)`. In SimSQL, a stochastic table can be annotated with a version number, which corresponds to a step of a Markov chain simulation or more generally to a level of recursion. In our case, `tableA[0]` denotes the initial version of `tableA`, and the SQL statement "`CREATE TABLE tableA[0] (aValue) AS`" in Figure 1 describes precisely how this table is to be created. Specifically, the VG function `Normal` is used to generate a temporary table `sampledVal`; the table-valued output of this VG function is then loaded into `tableA[0]` via the "`SELECT *`" subquery. The VG function `Normal` accepts a parameter table and outputs a result table of normally distributed samples.

Updating the parameters. The code to update σ^2, a and b—given in Figures 2 through 4—is similar to the code in Figure 1, with a few notable differences. First, the versioning variable `i` replaces the hard-coded `0`; this lets the SimSQL compiler know that this code is to be used to recursively define `variance`, `tableA` and `tableB`. Second, note the references to `variance[i-1]` in Figures 3 and 4, as well as the reference to `tableB[i-1]` in Figure 3. In this way, these stochastic table definitions reference

```
CREATE TABLE tableA[i] (aValue) AS
  WITH sampledVal AS SampleA (
    (SELECT * FROM VALUES (0, 10)),
    (SELECT 1 / (2 * var.s2Value)
     FROM   variance[i-1] as var),
    (SELECT SUM (2 * (y - bValue) * x), SUM (x * x)
     FROM   myTable, tableB[i-1]))
  SELECT sampledVal.value
  FROM   sampledVal;
```

Figure 3: Updating the value of a (σ_0^2 = 10).

```
CREATE TABLE tableB[i] (bValue) AS
  WITH sampledVal as Normal (
    (SELECT   SUM(y - aValue * x) /
              (var.s2Value / 10 + dataNum)
     FROM     variance[i-1] AS var, myData,
              tableA[i], metaData
     GROUP BY var.s2Value, dataNum),
    (SELECT   (1.0 / (1.0 / 10 + dataNum / var.s2Value))
     FROM     metaData, variance[i-1] AS var)
  )
  SELECT sampledVal.value
  FROM   sampledVal;
```

Figure 4: Updating the value of b (σ_0^2 = 10).

"older" versions of stochastic tables. Next, the code in Figure 3 uses the `SampleA` VG function. Since $P(a|.)$ has a non-standard distribution function, a library VG function cannot be used and the user-defined `SampleA` must be used instead; this VG function encapsulates a user-implemented "rejection" sampler [31] for $P(a|.)$. The three subqueries compute the statistics necessary to run the rejection sampler: the first (trivially) computes the parameter vector $(0, \sigma_0^2)$, the second computes $1/(2\sigma^2)$, and the third computes the quantities $\sum_i 2(y_i - bx_i)$ and $\sum_i x_i^2$ needed in the exponent that appears in the definition of $P(a|.)$. The situation is a bit simpler in Figures 2 and 4 in that the standard `InvGamma` and `Normal` VG functions are used to sample the updated values of σ^2 and b.

3. LATENT DIRICHLET ALLOCATION

We now turn to a more realistic example, and consider how SimSQL can be used to implement Bayesian inference for the popular LDA model [5]. LDA has many applications, from document classification to dimensionality reduction to document indexing. For reasons of exposition, we perform the inference using a "full" Gibbs sampler, as opposed to the "collapsed" sampler [29].

3.1 Model and Inference

LDA models a document as a bag of words where each word is generated by first selecting a latent (i.e., unobserved) topic from a document-specific topic distribution and then generating the word from a topic-specific word distribution. The topic distribution for each document is itself uncertain and has a Dirichlet prior. The Dirichlet family of "distributions over distributions" is the multivariate generalization of the beta distribution over $[0, 1]$; for $k \geq 2$ and parameter vector $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_k)$, the PDF is given by

$$\text{Dirichlet}(x_1, x_2, \ldots, x_k|\alpha) = \begin{cases} \frac{\Gamma\left(\sum_{i=1}^k \alpha_i\right)}{\prod_{i=1}^k \Gamma(\alpha_i)} \prod_{i=1}^k x_i^{\alpha_i - 1} & \text{if } x_i \geq 0 \text{ and } \sum_{i=1}^k x_i = 1; \\ 0 & \text{otherwise,} \end{cases}$$

where each α_i is positive and Γ is the standard gamma function. The word distribution for each topic is also uncertain, with a "symmetric" Dirichlet prior in which $\alpha_1 = \cdots = \alpha_k$. With a slight abuse of notation, we use a symbol such as α to denote both a

```
CREATE TABLE theta[0] (docID, topicID, prob) AS
    FOR EACH d IN documents
        WITH NewProbs AS Dirichlet
            (SELECT topicID, alpha
             FROM   topics, hyperparameters)
        SELECT d.docID, np.outID, np.probability
        FROM NewProbs AS np;
```

Figure 5: Initializing Θ

positive constant and the vector $(\alpha, \alpha, \ldots, \alpha)$, and simply write "Dirichlet(α)" to denote the corresponding symmetric prior.

Consider a large dataset of n documents with n_j words in the jth document and m distinct words overall. Assuming that t topics are present in the data and fixing hyperparameters α and β, the generative process underlying LDA is formally as follows:

1. For $i \in \{1...t\}$: $\Psi_i \sim$ Dirichlet(β)
2. For $j \in \{1...n\}$:
   (a) $\Theta_j \sim$ Dirichlet(α)
   (b) $z_j \sim$ Multinomial(n_j, Θ_j)
   (c) For $k \in \{1...t\}$: $w_{j,k} \sim$ Multinomial($z_{j,k}, \Psi_k$)

In step (1), we first produce a matrix Ψ with t rows and m columns; the row Ψ_i is a list of probabilities, where $\Psi_{i,d}$ is the probability that topic i will produce word d. Ψ_i is sampled from a Dirichlet(β) distribution. We subsequently use $\Psi_{*,d}$ to denote the column listing the probability that each topic will produce word d.

Next, we produce (for each document) a probability vector Θ_j, where $\Theta_{j,k}$ is the probability that a word in document j is generated from topic k. For document k, we sample z_j from a multinomial distribution with parameters n_j and Θ_j, so that $z_{j,k}$ records the number of words added to document j by topic k.

Finally, for document j, the quantity w_j is a matrix having t rows and m columns. $w_{j,k,d}$ tells us how many copies of word d were contributed to document j by topic k. Let $w_{j,k}$ denote the kth row in the w_j matrix, and let $w_{j,*,d}$ denote the dth column in this matrix. The kth row $w_{j,k}$ is sampled from a multinomial distribution with parameters $z_{j,k}$ and Ψ_k. After this process completes, the jth document in the corpus (represented as word frequencies) is $\delta_j = \sum_k w_{j,k}$. As discussed in Section 4, a typical goal of an LDA analysis is learning the posterior distribution of the Θ_j vectors, which describes the topic distribution within each document.

Our generative process is mathematically equivalent to the one described in the original LDA paper [5]; it differs slightly in that it is matrix- and vector-oriented, which leads to a simpler and more intuitive Gibbs sampler. It can be shown that the iterated sequence of updates required by the Gibbs sampler is as follows:

$$\Theta_j \sim \text{Dirichlet}(\alpha + z_j) \quad (2)$$

$$w_{j,*,d} \sim \text{Multinomial}(\delta_{j,d}, \Theta_j \times \Psi_{*,d}) \quad (3)$$

$$\Psi_i \sim \text{Dirichlet}(\beta + \sum_j w_{j,i}) \quad (4)$$

Here $\alpha + z_j$ denotes a vector addition and $\Theta_j \times \Psi_{*,d}$ denotes the item-by-item multiplication of the two vectors. We assume that the latter vector is normalized to a probability vector, if needed, prior to multinomial sampling. There is no need to explicitly sample z_j, because $z_{j,k}$ can always be computed as $\sum_d w_{j,k,d}$.

3.2 Specifying the Gibbs Sampler in SimSQL

To describe the above Gibbs sampler in SimSQL, we assume four database tables. The first two contain the topic and document identifiers, and the third contains a single row with the positive, real-valued Dirichlet hyperparameters α and β (recall our assumption of symmetric Dirichlet priors). The fourth table stores the number of times that each unique word appears in each document.

```
topics(topicID)
documents(docID)
hyperparameters(alpha,beta)    % hyperparameters
wordInDoc(docID,wordID,count) % words in documents
```

In addition, we have three stochastic tables; one (`psi`) to store the Ψ matrix, one (`theta`) to store the Θ matrix, and one (`w`) to store the various w matrices:

```
psi[i](topicID,wordID,prob)
theta[i](docID,topicID,prob)
w[i](docID,topicID,wordID,count)
```

Initialization. We begin by initializing the Θ and w matrices as shown in Figures 5 and 6. These codes are similar to the code in the previous section, with one big difference: the presence of the `FOR EACH` loop. Consider Figure 5. This code scans the `documents` table ("`FOR EACH d IN documents`") and, for each `d` encountered, invokes the `Dirichlet` VG Function via the `WITH` statement. The output schema for this VG function has two attributes: `outID`, the topic identifier, and `probability`, the word-production probability associated with that topic. The final `SELECT` query assembles the output from the VG function into a temporary table. All of the temporary tables created over all of the `d` values are then UNIONed to form `theta[0]`. Next, as shown in Figure 6, `w` is initialized by sampling, for each unique word in each document, the number of times that each topic produces the word, using the `Multinomial` VG function.

```
CREATE TABLE w[0] (docID, wordID, topicID, count) AS
    FOR EACH dw IN wordInDoc
        WITH TC AS Multinomial (
            (SELECT tm.topicID, tm.probability
             FROM theta[0] AS tm
             WHERE tm.docID = dw.docID),
            (SELECT dw.count))
        SELECT dw.docID, dw.wordID, tc.outID, tc.count
        FROM TC AS tc;
```

Figure 6: Initializing w

```
CREATE TABLE theta[i] (docID, topicID, prob) AS
    FOR EACH d IN documents
        WITH NewProbs AS Dirichlet
            (SELECT pw.topicID,
                    sum(pw.count) + hyperparameters.alpha
             FROM w[i-1] as pw, topics AS t, hyperparameters
             WHERE pw.docID = d.docID AND pw.topicID = t.topicID
             GROUP BY pw.topicID, hyperparameters.alpha)
        SELECT d.docID, np.outID, np.prob
        FROM NewProbs AS np;
```

Figure 7: Updating Θ

Updating the parameters. Figure 7 displays the SQL code for updating Θ. It is very similar to the initialization code in Figure 5; the key difference is in the parametrization of the `Dirichlet` VG function, which is based on the formula in (2). Similarly, the code to update `w`—shown in Figure 8—is very similar to the initialization code in Figure 6. Here the difference lies in the parametrization of the `Multinomial` VG function that allocates the number of appearances of a word in a document among the different topics. The parameter representing the probability that a given word in the document is associated with a given topic is computed as the product of the probability of the topic in the document and the probability that the topic produces the word, as required by (3).

```
CREATE TABLE w[i] (docID, wordID, topicID, count) AS
    FOR EACH dw IN wordInDoc
        WITH TC AS Multinomial(
            (SELECT tm.topicID, wpt.prob * tm.prob
             FROM psi[i-1] AS wpt, theta[i] AS tm
             WHERE wpt.wordID = dw.wordID AND
                   wpt.topicID = tm.topicID AND
                   tm.docID = dw.docID),
            (SELECT dw.count))
        SELECT dw.docID, dw.wordID, tc.outID, tc.count
        FROM TC AS tc;
```

Figure 8: Updating w

```
CREATE TABLE psi[i] (topicID, wordID, prob) AS
    FOR EACH t IN topics
        WITH NewProbs AS Dirichlet
            (SELECT pw.wordID,
                    sum(pw.count) + hyperparameters.beta
             FROM w[i] AS pw, hyperparameters
             WHERE pw.topicID = t.topicID
             GROUP BY pw.wordID, hyperparameters.beta)
        SELECT t.topicID, np.outID, np.probability
        FROM NewProbs AS np;
```

Figure 9: Updating Ψ

4. SPECIFYING AN ANALYSIS

We have shown how a database-valued Markov chain can be specified via a slight extension of standard SQL syntax. Denote by $D = (D[0], D[1], \ldots, D[M])$ the first $M + 1$ states of such a chain and recall that the overall goal of a SimSQL analysis is to study the (perhaps conditional) probability distribution of a query $Q = Q(D)$. (When using Gibbs sampling, the queries are usually of the specific form $Q(D[M])$, since interest centers on the state of the sampler after the burn-in period.) Here M is a maximum time-tick specified by the user—the current prototype requires that M be a fixed constant, but in future work we intend to relax this requirement. Conceptually, SimSQL first generates N samples from the distribution of D, i.e., N finite sample paths of the chain. SimSQL then applies Q to each of these sample paths to obtain a sample from the query-result distribution, and then uses these samples to estimate interesting features of the query-result distribution—such as moments, modes, and quantiles—or perhaps to visualize or perform statistical inference on the distribution.

In SimSQL, the user specifies an end-to-end simulation analysis using a novel SQL-like interface. Because of space constraints, we cannot describe the user interface in detail, and merely give a very brief example to show how a user might specify an analysis in the LDA scenario of Section 3. A typical goal of LDA is to study a quantity such as Θ_j, the (uncertain) topic probability vector for the jth document. Suppose that we wish to estimate $\tau_j = E[\Theta_j | w_j]$, the expected value of Θ_j, given the word data w_j. By the theory of Gibbs sampling, the sampled value $\Theta_j[M]$ after the Mth Gibbs iteration will be distributed approximately according to the distribution of $\Theta_j | w_j$ for M sufficiently large. Then we can estimate τ_j by $\hat{\tau}_j(N) = (1/N) \sum_{i=1}^{N} \Theta_j^{(i)}[M]$, where N is the number of Monte Carlo iterations (i.e., sample paths) and $\Theta_j^{(i)}[M]$ is the value of $\theta_j[M]$ in the ith iteration.

Suppose that $M = 50$ iterations is deemed large enough. Then, after defining the LDA model as in Section 3, the user can execute the program in Figure 10 to create a table of estimated topic-probability vectors, one per document. This table can then be further analyzed using standard SQL queries or other analysis tools. The `USING` clause specifies that $N = 100$ independent and identically distributed samples paths of the Gibbs-sampler Markov chain will be generated and used for subsequent averaging. The `WITH` clause specifies the query $Q(D)$ of interest. In this example, Q merely extracts the final (Mth) version of the `theta` table and returns this result as `lastTheta`. Note that the results from all N Monte Carlo iterations appear jointly in `lastTheta`. In particular, for any document j, the table contains the N sampled topic probability vectors $\Theta_j^{(1)}[M], \ldots, \Theta_j^{(N)}[M]$. The desired statistical summary of the Monte Carlo experiment is specified in the `COMPUTE` clause. For the LDA analysis, this summary is obtained simply by averaging the N Monte Carlo estimates of the topic probability, by document and topic.

```
USING IID(100)
WITH lastTheta(docID,topicID,prob,instanceID) AS
  (SELECT * FROM Theta[50])
COMPUTE
topicProbs(docID,topicID,prob) AS
  SELECT docID, topicID, AVERAGE(prob)
  FROM lastTheta
  GROUP BY docID, topicID
```

Figure 10: Syntax for an LDA analysis

5. IMPLEMENTATION OVERVIEW

The next few sections of the paper describe the SimSQL runtime. This section begins the description with an overview of the runtime.

Supporting huge plans. The defining characteristic of the MCMC simulations we would like to run in SimSQL is that the "query plans" are huge. A typical simulation may require tens to even hundreds of relational operations per iteration, so dozens of iterations may require thousands of operators.

One approach to running such a simulation would be to execute those thousands of operators as a single plan. However, there are problems with this approach. For example, cost-based optimization of such a huge plan is difficult because of the lack of reliable statistics over intermediate results. Estimating such statistics during optimization fails after the first few levels of the plan, due to the exponential propagation of the estimation errors [17].

A frame-based approach. We therefore perform incremental (re-) optimization and execution. Our approach is to partition the computation into sub-plans, called *frames*, which loosely correspond to the levels of recursive specification of the simulation. Each frame is optimized and executed in sequence: the output of one frame is provided as input to the next one. Dividing the computation into a sequence of frames provides an opportunity to collect precise statistics for each frame and use them when processing the next one. Most important, materializing the output of each frame makes it possible to checkpoint the computation. In addition to supporting recovery, checkpoints are crucial because they provide the ability to inspect intermediate states of a computation during a post-analysis phase. For example, imagine that after the completion of an 100 iteration MCMC process, a user wants to analyze the state of the Markov chain at the 20th iteration. Using a nearby checkpoint, the system can efficiently process that request.

Compiling and executing a simulation. The process, depicted in Figure 11, is as follows:

1. The SimSQL compiler parses the specification and creates a number of *template plans* that will be strung together to form the simulation. Template plans are so-named because they are the unoptimized building blocks from which a frame is constructed—they serve as templates for various parts of the frame. At a minimum, the system creates a template plan for each SQL code snippet that specifies the initial version

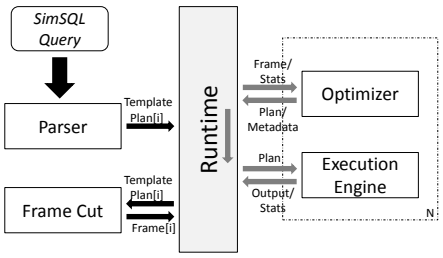


Figure 11: Overview of query processing in SimSQL.

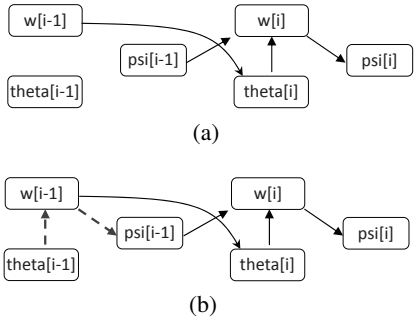


Figure 12: Determining whether a recursive query plan is valid.

of a stochastic table, and a template plan for each SQL code snippet that defines an intermediate stochastic table.

2. In a cost-based fashion, the set of template plans defining the intermediate tables are partitioned via a "frame cut" into two subsets. This partitioning determines where one frame ends and another begins, and is done so as to minimize the amount of data that needs to be materialized at the end of each frame.

3. The template plans for the initialization are appended with all of the template plans up to and including those that are part of the cut. This is the first frame, which is optimized.

4. After optimization, the deterministic parts of the first frame are identified, and the frame is run. As the frame is run, the deterministic parts are materialized.

5. While the result of running the frame is written out, statistics describing the resulting output tables are computed.

6. The next frame is created and optimized. The optimizer has the option of deciding (in cost-based fashion) to integrate the materialized, deterministic "views" from Step 4 into the plan for the frame.

7. The resulting plan for the frame is run by the SimSQL execution engine.

8. Steps (5) through (7) are repeated until completion.

6. QUERY PROCESSING IN DETAIL

This section describes various aspects of the aforementioned query processing steps in detail. A discussion of step (7) above—running a query plan—is deferred until the next full section of the paper.

6.1 Compilation

The SimSQL compiler performs multiple tasks: semantic checking of the SQL, flattening of nested queries, and generating template and actual plans.

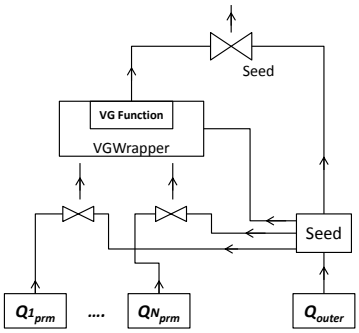


Figure 13: Template plan for creating a random table.

Checking for valid recursive definitions. The semantic checks that are performed by the query compiler are fairly standard, with an additional check for correctness of recursive table definitions. This latter check ensures that (1) the schema of a stochastic table `A[i]` is consistent, (2) each stochastic table has exactly one recursive definition, (3) a stochastic table with a constant index `A[const]` does not reference a stochastic table with a varying index `B[i]` and (4) at least one stochastic table with a constant index is defined. The current implementation assumes that a stochastic table `A` with index `i` can only reference another stochastic table `B` with index `i-1` or `i`.

The system also ensures that every stochastic table can be derived from stochastic tables with constant indices:

1. First a graph is generated whose nodes are the stochastic tables with indexes `i` and `i-1`, and the edges represent the dependence relationships among them. See, for example, Figure 12(a) for LDA.

2. As in Figure 12(b), edges among stochastic tables with index `i` are copied to the corresponding tables with index `i-1`.

3. If all the stochastic tables with index `i` can be accessed, then this check is passed.

Compilation into template plans. A template plan is simply an executable, unoptimized query plan. There are three types of template plans: *baseline*, *intermediate*, and *branch* template plans. Baseline plans correspond to stochastic table definitions with constant indices. In the LDA example, `theta[0]` and `w[0]` are compiled into baseline plans. Intermediate plans correspond to stochastic table definitions that recursively depend upon other stochastic tables, and are themselves used to parametrize other stochastic tables. In the LDA example, the `theta[i]`, `w[i]`, and `psi[i]` definitions are compiled into three intermediate template plans. Figure 14 shows the four template plans for the LDA example. Branch plans correspond to SQL queries that are run on top of stochastic tables to extract data or perform analysis.

Compiling a stochastic table definition into a template plan requires some explanation. The template plan for a generic stochastic table definition shown in Figure 13. Note that this template plan replaces the MCDB `Instantiate` operator, and is preferable in that it is an actual plan that is open to algebraic manipulation by the optimizer. In the figure, the plans $Q1_{prm}$ to QN_{prm} instantiate the tables that are used to parametrize the VG function, and Q_{outer} the "outer table" that corresponds to the tuples that are iterated over in the `FOR EACH` statement of a stochastic table definition. The `Seed` operator appends an integer (the "seed"), to every tuple from

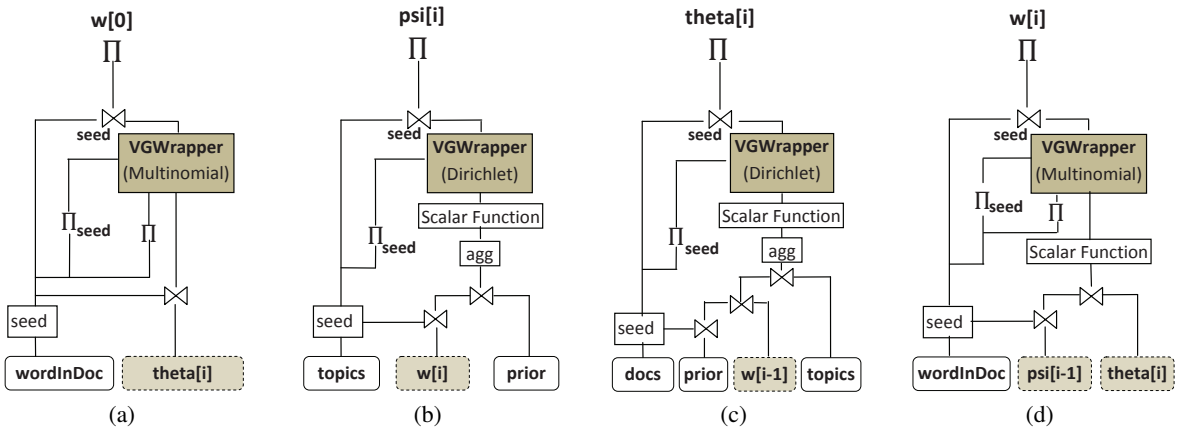


Figure 14: Template plans for LDA

Q_{outer}. The bottom level joins in the plan add seeds to each parameter stream based on the predicate in the `WHERE` clause of the SQL used to define the corresponding parameter tables. Seeds play three different roles in SimSQL. First, they seed the pseudorandom number generators (PRNGs) used within VG functions. Second, they are used as keys to match tuples from parameter streams with corresponding tuples in the outer table stream. Finally, seeds are used to link the output tuples of the VG function that correspond to a tuple in the outer table stream in the top level join operation. The actual VG function is run by the `VGWrapper` operator. It is described in detail in Section 7.3.

6.2 Frame Cut

As the execution of a frame is completed by SimSQL, the system checkpoints the state of the computation by writing out and saving the set of stochastic tables that will be used to parametrize the next frame. If this is done naively and very large tables parametrize the next frame, the size of the checkpoint may be huge. Thus, the SimSQL system chooses to end each frame at a "narrow" point in the computation, where the stochastic tables needed to parametrize the next frame take little space. The following algorithm computes the smallest checkpoint possible:

1. Given the template plans, a logical plan with two iterations is generated. This logical plan is optimized and executed to obtain statistics for each stochastic table.
2. An acyclic directed graph $G = (V, E)$ is defined, where V consists of stochastic tables with indices `i-1` and `i`, and E encodes the dependencies among them.
3. The *source* and *sink* are added. For each table `A` $\in V$, if `A` references tables with indices `i-2`, a direct edge from *source* to `A` is added; similarly, if `A` is referenced by tables with indices `i+1`, a directed edge from `A` to *sink* is added.
4. We assign the weight $+\infty$ to each edge $e \in E$ and the weight (cost) from Step 1 to each node $v \in V$. Our problem is then transformed to a max-flow/min-cut problem [11] with vertex capacities, and we apply the Edmonds-Karp algorithm.

Figure 15 shows the graph used to compute the LDA frame cut. The assigned weights (costs for materialization) are displayed for each stochastic table. After applying the Edmonds-Karp algorithm to compute the minimum cut, we find (as expected) the optimal frame cut is at {`psi[i-1]`,`theta[i]`}. Thus, these two tables will be checkpointed and will serve as input into every frame.

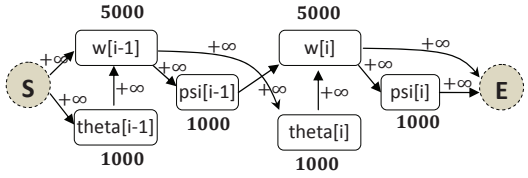


Figure 15: The graph used to determine the optimal frame cut.

6.3 Building/Optimizing Frames

The first frame is produced by starting with the template plans for the baseline tables, and then attaching (as specified by the SQL) all subsequent template plans up to and including those plans in the optimal cut, for `i` $= 0$. Subsequent frames are produced by assuming the results of the optimal cut as input, and then including all of the template plans in a full iteration. An example frame for LDA is shown in Figure 16.

This frame is then passed to the SimSQL query optimizer. Unfortunately, space precludes us from describing the extended SimSQL's query optimizer in detail. Briefly, the optimizer uses an A*-inspired search to optimize the plan. The cost of a plan is a weighted, linear combination of the total number of bytes input to and output from each intermediate plan operator. The weights on the various operator types capture the importance of each operator in the query cost estimation and are calculated using linear regression. SimSQL's optimizer has at its disposal all of the usual transformations (such as join re-orderings and push-down of selections) and also a list of SimSQL-specific transformations. An example of an SimSQL transformation is shown in Figure 17, where a random attribute called `count` is dropped and later re-created using the seed. Because random attributes are implemented as an array of random values, carrying them around can be quite expensive; thus dropping and re-creating a random attribute may result in a less expensive plan.

6.4 Materializing Deterministic Sub-Plans

The optimized frame may contain *deterministic sub-plans*, which are sub-plans of the query plan containing no `VGWrapper` operations, and where the initial input comes only from deterministic database tables. The output from such a plan can be re-used. To identify the deterministic sub-plans we employ a simple bottom-up algorithm. The algorithm visits each operator based on its topological ordering. An operator belongs to a deterministic plan if (1) it

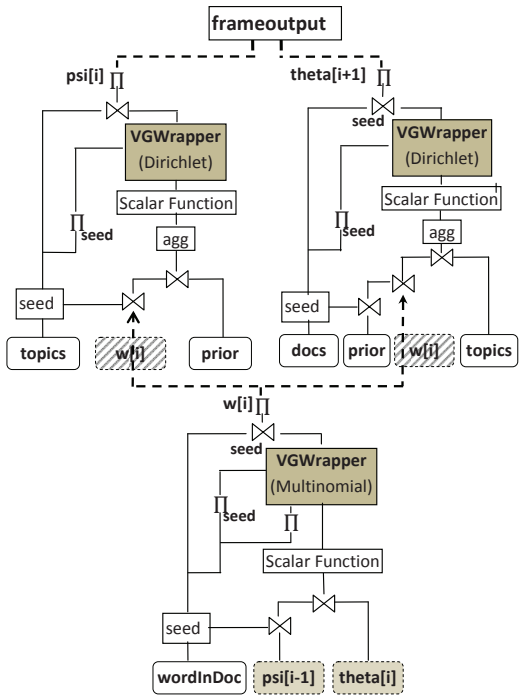


Figure 16: Frame to be optimized for execution.

is a deterministic database table scan or (2) all its descendants belong to a deterministic plan. Following standard methods used to incorporate materialized views into query plans [15], we give the optimizer the opportunity of re-using a deterministic sub-plan by introducing a plan transformation.

7. SIMSQL EXECUTION ENGINE

This section of the paper describes various aspects of the SimSQL execution engine.

7.1 Overview

The SimSQL execution engine is built on top of Hadoop, and written in both Java and C++. The ultimate output for the SimSQL compiler and optimizer (and the input into the SimSQL execution engine) is a specification for the query plan to be run, in a SimSQL-specific dataflow language that describes all of the relational operations in the plan, as well as how they fit together.

The SimSQL execution engine compiles this plan into a set of MapReduce jobs, and then, taking into account the dependencies among them, it runs them one at a time. To run a particular MapReduce job, SimSQL first "writes" (using a special macro-expansion facility) a Java code for the desired map task and a Java code for the the desired reduce task. These Java codes contain logic that is specifically tailored to the task at hand. For example, if the goal is to execute a relational join, the generated Java might contain (on the map side) specially generated code to run selection predicates over the two input relations, project away un-needed attributes, and compute hash keys over the input records. The generated Java code for the reduce side might contain code for the join predicate, as well as code for the final projection and any arithmetic or other functions that must be run to produce the final output.

Once the Java code for the task is generated, it is compiled and then executed. The reason that we employ this sort of dynamic code generation is to bypass the high CPU cost associated with interpreting, at runtime, data structures describing how to process the records dynamically; instead, we allow an optimizing compiler to generate byte code that does this far more efficiently. Others have explored this idea in the past few years (see, for example, the work of Neumann [26]). While such dynamic code generation has a cost—namely, the time required to generate and compile the code—in a MapReduce environment the additional fraction of a second is a small price to pay when one considers that the fastest MapReduce jobs take tens of seconds to run.

7.2 Tuple Bundles

Unlike in most applications of Hadoop, the data processed by SimSQL conform to a SimSQL-specific binary format instead of the usual text-based format. The important point to note is that the SimSQL format incorporates the MCDB "tuple bundle" trick [18]. Conceptually, SimSQL repeats the desired simulation N times to produce N mutually independent runs; in fact, the simulation is executed only once. This is made possible through SimSQL's use of *tuple bundles*, a concept borrowed from MCDB. Rather than running the simulation N times, a single set of tuple bundles that encodes each of the N independent possible worlds flows through a single query plan.

A tuple bundle t with schema S is, logically speaking, an array of N tuples, all having schema S. An attribute `att` is *constant* if $t[i]$.`att` $= c$ for some fixed value c and $i = 1, 2, \ldots, N$, otherwise it is *stochastic*. In our SimSQL implementation, the N tuples making up a single tuple bundle are stored in the obvious way: constant attributes are represented using a singleton value, and stochastic attributes are stored as arrays of values. A special stochastic attribute called `isPres` may also be present in the tuple bundle. The value of $t[i]$.`isPres` equals $true$ if and only if the tuple bundle has a constituent tuple that appears in the ith possible world.

7.3 The VG Wrapper Operation

Most of SimSQL's MapReduce implementations of relational operations (the various joins such as the natural join, anti-join, and semi-join, de-duplication, selection, and so on) are straightforward. However, one operation that deserves a bit more explanation is SimSQL's `VGWrapper` operation. Each `VGWrapper` operation is run as part of a template plan, as shown in Figure 13 (this was discussed in the previous section), and the `VGWrapper` is tasked with parameterizing and executing a particular VG function.

Every VG function in SimSQL is a C++ class with (at least) four methods: `ClearParams`, `TakeParams`, `TakeSeed`, and `OutputVals`. `ClearParams` is called to let the VG function know that it is about to be parameterized. After `ClearParams`, `TakeParams` is called repeatedly to parameterize the VG function. When the `VGWrapper` is ready to use the function to produce random data, `TakeSeed` is called with the seed for the VG function's PRNG. The `VGWrapper` then repeatedly calls `OutputVals` to obtain random values from the VG function. When a `null` is eventually returned by the VG function (indicating that it is done producing values for the current possible world), the `VGWrapper` will then (optionally) call `ClearParams` to indicate that it wants to re-parameterize the function for the next trial (if the parameterization is constant across worlds, then no re-parameterization is needed). The `VGWrapper` then calls `TakeSeed`, followed by a sequence of calls to `OutputVals`, until a `null` is once again returned. This process is completed until all worlds have been computed.

To demonstrate this via an example, consider a `Multinomial` VG function and the `w[i]` stochastic table from the previous LDA

example. Imagine that there are four tuple bundles, encoding three possible worlds, that are used to parameterize the function:

(tID: 0, prob: ⟨ 0.1, 0.2, 0.1 ⟩, isPres: ⟨ T, T, T ⟩)
(tID: 1, prob: ⟨ 0.4, 0.6, 0.5 ⟩, isPres: ⟨ F, T, F ⟩)
(tID: 2, prob: ⟨ 0.5, 0.2, 0.4 ⟩, isPres: ⟨ T, F, T ⟩)
(count: 14)

Here, a ⟨ ⟩ pair indicates an array of values, with each value being applicable to one of the three possible worlds. In our example, the VGWrapper will begin with the first possible world, calling ClearParams to indicate the start of a parameterization, followed by three calls to TakeParams, giving (0, 0.1), (2, 0.5), and (14) in sequence. The VGWrapper then calls TakeSeed, passing along the seed associated with the current dw value (see Figure 8) from the outer table stream (Q_{outer} in Figure 13). The VGWrapper then repeatedly calls OutputVals to obtain all output tuples for the first possible world. The VGWrapper then calls ClearParams and parameterizes the second possible world with (0, 0.2), (1, 0.6), and (14), and obtains the result with calls to OutputVals. Finally, it parameterizes the third possible world with (0, 0.1), (2, 0.4), and (14), obtains the result.

Note that we made the somewhat non-obvious decision to re-seed the VG function's PRNG for each and every possible world. This is in contrast to MCDB, where the VG function is seeded only once, and then the seed is used for all of the possible worlds. The reason that SimSQL re-seeds for each possible world is so that we can handle the presence of the isPres attribute in the outer input table with some efficiency. The stream of tuples from the outer input table supplies the seeded FOR EACH tuples to the VGWrapper. If one of these tuples has a false value in isPres, it means that the outer tuple has been removed from the database (perhaps via an earlier selection) and logically, it should not exist. A simple way to handle the non-existence would be to have the VGWrapper ignore it, knowing that it will be enforced by the top-level join in Figure 13. Although this solution is correct, it might be costly, since the VG function will be invoked needlessly.

Another solution would be to avoid running the VG function for possible worlds where isPres is false. This must be done with care. As discussed in the previous section, the optimizer is free to drop (project away) random attributes, and then generate them again later via a second execution of the VGWrapper operator. This transformation is depicted in Figure 17, and is useful if carrying around random data is more expensive than re-generating it. If the VGWrapper is invoked once, and then a subsequent operation removes an outer stream tuple from a possible world, the sequence of calls to the VG function will change the second time around, because that possible world will not be parameterized nor will OutputVals be called for that possible world. If the VG function was parameterized only once, at initialization, then the second use of the VGWrapper will produce different, inconsistent results because it will not be run for certain possible worlds.

Thus, in SimSQL the seed associated with a tuple from the outer table stream is not given directly to the VG function. Rather, it is used to seed an additional PRNG that is internal to the VGWrapper, and which is used to re-seed the VG function for each possible world via a call to TakeSeed. If isPres is false for any possible world, the new seed is discarded. That way, a possible world is always assigned the same seed, regardless of changes to the isPres attribute.

8. EVALUATION

In this section, we empirically evaluate the SimSQL system. Our goal is not to show that SimSQL can be used to implement the fastest distributed MCMC simulation possible—it would be unreasonable to think that a simulation coded using 47 lines of SQL (as with LDA) could possibly outperform a distributed simulation custom coded in Java or C. Rather, we wish to give some evidence that a simulation implemented and run on the SimSQL platform can have reasonable performance. "Reasonable" here might be a time that is within an order of magnitude (or two) of a special-purpose Java or C version. A $10X$ slowdown might be a big performance hit, but a distributed Java or C version might require several hundred times as much code, much more development time, and a much higher level of programming expertise.

8.1 Overview

Most of SimSQL as described in the paper has been implemented, with the exception of the analysis interface of Section 4. SimSQL consists of an SQL query compiler and query un-nester (approximately 40,000 lines of Java code), a cost-based query optimizer (approximately 11,000 lines of Prolog code), and a runtime (written in around 30,000 lines of Java and C/C++) that is built upon Hadoop. We use Amazon EC2 for all of our runs. The machines used are "High-Memory Quadruple Extra Large Instance" types, which have 8 relatively low-end virtual cores, two disks, and 68GB of RAM per machine.

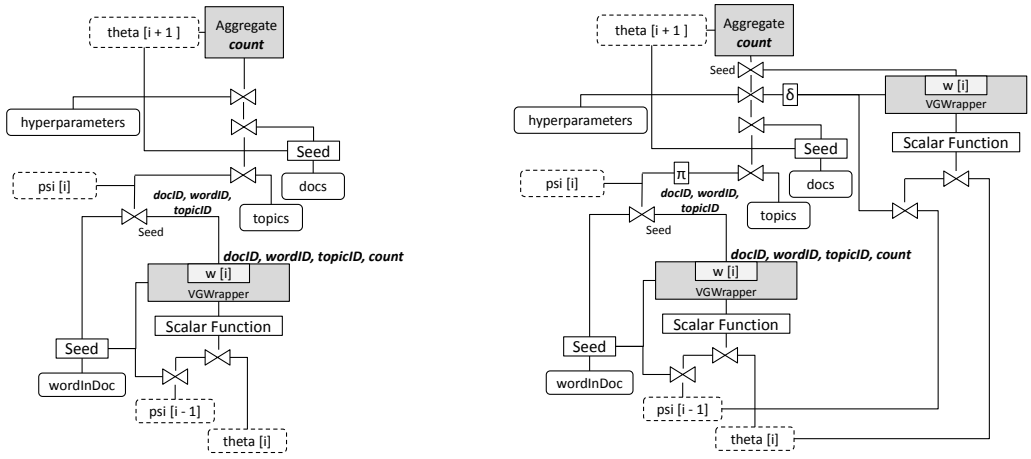
Our high-level goal is examining SimSQL's feasibility. Along these lines, we wish to answer the questions: How well does SimSQL scale as the amount of data and computing resources increases? Can SimSQL run a distributed simulation quickly and (even more importantly) in a cost effective manner? To answer this question, we perform an experiment where the amount of data per compute node in a cluster is held constant, but an ever-increasing number of nodes are used to perform the simulation. If SimSQL scales perfectly, the time to run each simulation will stay constant as the number of nodes is increased.

We focus on two particular simulations: the MCMC simulation for the LDA model of Section 3 and a more complicated MCMC simulation for learning a Bayesian Gaussian mixture model (GMM) [4]. Unfortunately, space precludes us from providing the corresponding SQL code, but it totals 63 lines, and provides support for full (not just diagonal) covariance matrices. The LDA SQL code listed in the paper is compiled and run without modification. The LDA simulation compiles to a query plan with 45 operations in the baseline iteration, and 39 operations in the other iterations (that run in 22 and 20 MapReduce jobs, respectively). The GMM simulation has 40 operations in the baseline iteration, and 87 operations in the other iterations (running in 20 and 36 MapReduce jobs).

For the LDA model, we use the 20-newsgroups dataset [1] in our experiments. Since this dataset has only 16,330 documents, we create a database of (almost) arbitrary size by randomly picking two documents from the corpus and merging them—this gives us $\binom{2\times 16330}{2}$ distinct "documents". While this corpus of semi-simulated documents may not be the best testbed for evaluating the LDA model itself, we are interested in evaluating the SimSQL system, and this data should suffice. In our experiments, we use 100 topics and a dictionary of 10,000 words. For the GMM model, we use synthetically-generated ten dimensional data. The data are themselves generated by sampling from a Gaussian mixture model.

8.2 Benchmarks Run

Scalability. For the two models, we first choose a per-machine data size n, such that across experiments, when performing a simulation using m machines, the total amount of data processed is always $n \times m$, no matter the number of nodes. In the case of LDA, n is 250,000 documents. In the case of the GMM, n is 1,000,000 data



(a) Plan before transformation.

(b) Plan after transformation.

Figure 17: Transformation to drop and recreate a random attribute in SimSQL.

LDA Inference

Nodes	Iteration Number					
	0	1	2	3	4	5
5	2:22	2:33	2:28	2:25	2:29	2:22
10	2:33	2:35	2:30	2:28	2:32	2:30
20	2:34	2:27	2:30	2:28	2:30	2:34
50	2:44	2:39	2:37	2:41	2:52	2:44
100	3:24	3:16	3:17	3:18	3:13	3:04

Gaussian Mixture Model Inference

Nodes	Iteration Number					
	0	1	2	3	4	5
5	0:06	0:54	0:53	0:54	0:53	0:55
10	0:06	0:55	0:53	0:53	0:52	0:53
20	0:06	0:57	0:56	0:57	0:56	0:56
50	0:08	1:10	1:07	1:04	1:01	1:02
100	0:08	1:13	1:13	1:13	1:12	1:12

Figure 18: Running time (HH:MM) per iteration for GMM and LDA inference. Iteration zero is the initialization.

points. For each run, 10 different simulations are performed at the same time, so that 10 parallel Markov chains are simulated. We run the simulations for the two models using 5 nodes, 10 nodes, 20 nodes, 50 nodes, and 100 nodes. Figure 18 displays the results.

Work Sharing. To test the ability of the system to share work across possible worlds, we fix the number of nodes at 20. Then we vary n as follows. Let N be the number of Markov chains that are simulated in parallel. For LDA, we fix $n \times N$ to be $20 \times 10 \times 250,000$, and for the GMM, we fix $n \times N$ to be $20 \times 10 \times 1,000,000$, and then we perform five runs of the simulation, using $N = \{1, 5, 10, 20\}$. Thus, if $N = 10$, the run is identical to the scalability experiment with $m = 20$. But with $N = 20$, half the amount of data per machine is used, and if $N = 1$, ten times the amount of data per machine is used. Figure 19 gives the results.

Model	Number of parallel worlds			
	20	10	5	1
LDA	2:11	2:28	3:27	4:20
GMM	0:51	0:57	1:27	1:53

Figure 19: Avg running time per iteration (HH:MM), changing the number of parallel simulations while keeping the (number of data points) $\times$ (the number of simulations) constant.

8.3 Discussion

The system does appear to scale. The running time is more or less constant when the total data processed is held constant at $n \times m$. That said, not all of the rows in Figure 18 are identical for differing cluster sizes. Much of the increase in time is due to the fact that as the size of the problem increases, it becomes impossible for the system to pipeline expensive operations, such as joins. For example, in our Amazon cluster, we have approximately 6GB of RAM available for operation-specific, auxiliary data structures, for each Hadoop mapper and reducer. In the case of the GMM simulation, as we move from 10 to 20 machines (and 10 million to 20 million data points) it becomes impossible to buffer a copy of the entire data set in a 6GB hash table to run a join, and so certain joins switch from single-pass pipelined joins to full sort-merge joins, and the time increases slightly. As we move from 50 to 100 machines, we find that it is not always possible to store meta-data about all 100 million data points (such as the PRNG seed assigned to each) in RAM, and again we see an increase in running time. That said, for both GMM and LDA, at 100 machines, none of the varying-sized tables can be buffered in RAM, and so we would expect a more constant running time as the cluster size grows even larger.

Although the system appears to scale, it is reasonable to ask whether a running time of two or three hours per iteration is practical. To many practitioners of Bayesian machine learning, who have long been used to month-long simulation runs, the answer is affirmative. Additionally, we assert that the absolute time taken per iteration is perhaps not that important for a cloud-based platform. If the underlying architecture scales, the running time can be driven down dramatically at no additional cost by simply renting more machines. Specifically, our 100-node cluster costs around $14 per hour (14 cents per machine per hour), and can run a GMM iteration in about 70 minutes—or a bit more than $15 per iteration. We could instead rent 200 machines, at a cost of $28 per hour, and as long as the running time per iteration is cut approximately in half, we will

still spend $15 per iteration, but would now have to wait only half as long for each iteration to finish. True, increasing cluster size past a certain point produces diminishing returns. In our Hadoop-based system it is impossible to push the per-iteration running time down below a few minutes (each GMM iteration requires 36 MapReduce jobs, which will always take at least seconds each, even in a small cluster). But in general, if the simulation takes too long, one can just rent a larger cluster.

Perhaps a more reasonable question is: Is the *cost* of learning a model reasonable? First, consider how many iterations are required for convergence. Burn-in periods of 1,000 or more iterations are common in the machine learning literature, but we question whether this is always a necessity. All of the standard Bayesian models with which we are familiar—including LDA, GMM, linear regression, conditional random fields, and matrix factorization—become quite stable after 100 iterations or so. For example, in Figure 20 we plot the θ vectors obtained in iterations 50-60 for three different documents from a run of LDA on the 20-newsgroups data set. There is little fluctuation, indicating stability in the model.

If we assume 100 iterations are desired, then at $14 per iteration to handle 100 million data points (in the case of the GMM simulation), we have a cost of $14 per million data points. Considering that we are running 10 simulations in parallel at that cost,[1] and that running a single simulation cuts the cost by a factor of five (see Table 19), we could run a single MCMC inference at the cost of around $3 per million data points.

This is certainly not inexpensive, perhaps $10\times$ to $100\times$ more expensive than what one would obtain with a special-purpose, C or Java implementation, such as the LDA code written by Smola and Narayanamurthy [33]. That said, programmers who can write high-quality distributed or parallel C/Java code are very expensive themselves (and quite rare!), and so we argue that the fact that we obtain a cost of $3 per million data points with only 63 lines of SQL makes our approach look very attractive—especially when the goal is fast prototyping of a particular inference algorithm.

9. RELATED WORK

Perhaps the work closest to our own in the database literature is the paper of Deutsch, Koch, and Milo on languages for specifying queries over Markov chains [13]. In contrast to our systems-oriented approach where the goal is to prototype a distributed platform suitable for performing large scale Bayesian MCMC, Deutch et al. are concerned with algorithmic aspects of the problem, showing how in certain cases it is possible to answer queries over a Markov chain without needing to actually simulate the chain many times. The setting that they consider is somewhat restrictive compared to our own (probabilistic databases with c-tables [16], or repair-keys [21]) and it is not immediately apparent how this work applies to common distributions such as the Dirichlet that accept real, vector-valued inputs and produce correlated, real, vector-valued outputs. However, there is a lot of merit to the development of algorithms to solve this problem efficiently, and the ultimate solution will include novel algorithmic as well as systems-oriented ideas.

MCMC for model inference over database data has been previously considered [36, 38]. In [36], the MCMC loop takes place within an external module. In [38] efficient SQL-based implementations of two MCMC algorithms were described. However, the work is not meant to provide general-purpose MCMC simulation.

[1]The advantage of running many simulations at the same time is that it generates many fully-independent models, without the need to continue to run a single chain, examining its state occasionally, to obtain more than one learned model.

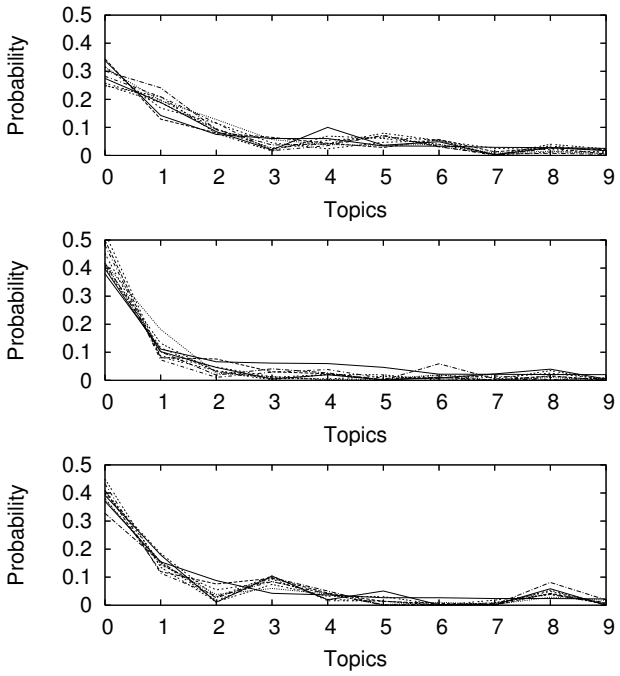


Figure 20: Parallel coordinates plot depicting sampled LDA θ vectors for three documents, iterations 50-60. The top ten topics for each document are shown, sorted in order of importance.

Jigsaw [20] is a system that uses VG functions to perform stochastic, what-if analysis. Jigsaw focuses not on systems issues (language, compilation, runtime), but instead on the problem of optimization over stochastic models; most of the technical innovation in the Jigsaw system is directed towards alleviating Monte Carlo overheads by decreasing the number of VG function invocations required when a stochastic simulation must be repeated for many different parameter configurations during the optimization.

Recent work in the machine learning and data mining communities has leveraged the MapReduce paradigm to either achieve high levels of parallelism [10, 24] or scale specific machine learning algorithms [3, 22, 23, 25, 27, 28, 32, 33]. These approaches require the detailed implementation of the model in a procedural language.

The SimSQL implementation described in this paper compiles simulations into MapReduce programs. The MapReduce framework has been employed to scale up other analytic tasks [12, 30].

The sequential execution of small frames in SimSQL is related to the repeated computation that is supported by the Pegasus [19], SystemML [14] and HaLoop [6] systems, though our emphasis on checkpointing and (re-)optimization is unique. SimSQL is closest in spirit to Haloop, which extends the MapReduce framework to efficiently support simple iterative or recursive computations.

A number of systems support SQL or SQL-inspired languages on top of MapReduce. These include DryadLINQ [37], SCOPE [7], HadoopDB [2], Hive [34], FlumeJava [8], and Tenzing [9].

Finally, there exists some recent work on scaling up specific classes of simulation using MapReduce, for example, agent-based simulations [35] and clustering for N-body simulations [22].

10. CONCLUSION

We have described the functionality, design and implementation of the SimSQL system, which allows for SQL-based specification, simulation, and querying of database-valued Markov chains. The

key advantage of the SimSQL system is that allows a programmer to specify Markovian simulations of relatively high complexity in only a few dozen lines of SQL code; these simulations are then automatically distributed and run in parallel over a cluster of machines. As a proof-of-concept we described in detail the SQL specification of two real world Bayesian inference algorithms: MCMC inference for linear regression and for LDA. We also discussed possible SQL specifications of analysis tasks over the results of the performed simulation. Our experiments, using the LDA and GMM models, show that SimSQL has reasonable performance and scales well when running distributed Markov chain simulations.

11. REFERENCES

[1] 20Newsgroups Dataset. Available at: http://people.csail.mit.edu/jrennie/20newsgroups/.

[2] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *PVLDB*, 2(1):922–933, 2009.

[3] B. Bahmani, K. Chakrabarti, and D. Xin. Fast Personalized Pagerank on MapReduce. In *SIGMOD*, 2011.

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

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

[6] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: efficient iterative data processing on large clusters. *PVLDB*, 3(1-2):285–296, 2010.

[7] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2):1265–1276, 2008.

[8] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *PLDA*, 2010.

[9] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, and M. Wong. Tenzing a sql implementation on the mapreduce framework. *PVLDB*, 4(12):1318–1327, 2011.

[10] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *NIPS*, 2006.

[11] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill, 2003.

[12] S. Das, Y. Sismanis, K. S. Beyer, R. Gemulla, P. J. Haas, and J. McPherson. Ricardo: Integrating r and hadoop. In *SIGMOD*, 2010.

[13] D. Deutch, C. Koch, and T. Milo. On probabilistic fixpoint and markov chain query languages. In *PODS*, pages 215–226, 2010.

[14] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative machine learning on mapreduce. In *ICDE*, 2011.

[15] J. Goldstein and P. Larson. Optimizing Queries Using Materialized Views: A practical, scalable solution. In *SIGMOD*, 2001.

[16] T. J. Green and V. Tannen. Models for incomplete and probabilistic information. *IEEE Data Eng. Bull.*, 29(1):17–24, 2006.

[17] Y. E. Ioannidis and S. Christodoulakis. Optimal histograms for limiting worst-case error propagation in the size of join results. *TODS*, 18(4):709–748, 1993.

[18] 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.

[19] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system - implementation and observations. In *ICDM*, 2009.

[20] O. Kennedy and S. Nath. Jigsaw: Efficient optimization over uncertain enterprise data. In *SIGMOD*, 2011.

[21] C. Koch. On query algebras for probabilistic databases. *SIGMOD Record*, 37(4):78–85, 2008.

[22] Y. Kwon, D. Nunley, J. D. Gardner, M. Balazinska, B. Howe, and S. Loebman. Scalable clustering for n-body simulations in a shared-nothing cluster. In *SSDBM*, 2010.

[23] Z. Liu, Y. Zhang, E. Y. Chang, and M. Sun. Plda+: Parallel Latent Dirichlet Allocation with Data Placement and Pipeline Processing. *ACM TIST*, 2(3):26:1–26:18, 2011.

[24] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A New Parallel Framework for Machine Learning. In *UAI*, 2010.

[25] A. Mahout. Available at: http://mahout.apache.org/.

[26] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[27] B. Panda, J. S. Herbach, S. Basu, and R. J. Bayardo. Planet: Massively parallel learning of tree ensembles with mapreduce. *PVLDB*, 2(2):1426–1437, 2009.

[28] S. Papadimitriou and J. Sun. DisCo: Distributed co-clustering with map-reduce: A case study towards petabyte-scale end-to-end mining. In *ICDM*, 2008.

[29] I. Porteous, D. Newman, A. T. Ihler, A. Asuncion, P. Smyth, and M. Welling. Fast collapsed Gibbs sampling for Latent Dirichlet Allocation. In *SIGKDD*, 2008.

[30] RHIPE: R and Hadoop Integrated Programming Environment. Available at: http://ml.stat.purdue.edu/rhipe/.

[31] C. P. Robert and G. Casella. *Monte Carlo Statistical Methods*. Springer, 2010.

[32] S. Singh, A. Subramanya, F. Pereira, and A. McCallum. Distributed MAP inference for undirected graphical models. In *NIPS Workshops*, 2010.

[33] A. J. Smola and S. Narayanamurthy. An Architecture for Parallel Topic models. *PVLDB*, 3(1):703–710, 2010.

[34] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[35] G. Wang, M. V. Salles, B. Sowell, X. Wang, T. Cao, A. Demers, J. Gehrke, and W. White. Behavioral simulations in mapreduce. *PVLDB*, 3(1-2):952–963, 2010.

[36] M. Wick, A. McCallum, and G. Miklau. Scalable probabilistic databases with factor graphs and MCMC. In *VLDB*, 2010.

[37] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

[38] D. Z. Zhang, M. J. Franklin, M. Garofalakis, J. M. Hellerstein, and M. L. Wick. Hybrid in-database inference for declarative information extraction. In *SIGMOD*, 2011.