The BUDS Language for Distributed Bayesian Machine Learning

Zekai J. Gao, Shangyu Luo, Luis L. Perez, Chris Jermaine
Rice University
Houston, TX
{jacobgao, sl45, lp6, cmj4}@rice.edu

ABSTRACT

We describe BUDS, a declarative language for succinctly and simply specifying the implementation of large-scale machine learning algorithms on a distributed computing platform. The types supported in BUDS–vectors, arrays, etc.–are simply logical abstractions useful for programming, and do not correspond to the actual implementation. In fact, BUDS automatically chooses the physical realization of these abstractions in a distributed system, by taking into account the characteristics of the data. Likewise, there are many available implementations of the abstract operations offered by BUDS (matrix multiplies, transposes, Hadamard products, etc.). These are tightly coupled with the physical representation. In BUDS, these implementations are co-optimized along with the representation. All of this allows for the BUDS compiler to automatically perform deep optimizations of the user's program, and automatically generate efficient implementations.

1. INTRODUCTION

Developing a statistical or machine learning (ML) application that derives value from a large data set is difficult. Unless the application uses a standard model whose usage is widely understood and for which a library implementation exists (such as logistic regression or k-means clustering), building such an application typically requires a three-step process:

1. **The whiteboard step**–Do the math to define the model and derive the learning algorithm.
2. **The small data prototype**–Build a prototype of the model/learning algorithm using a tool such as Matlab or R, and evaluate the prototype using a sub-sample of the data.
3. **The big data deployment**–Build a robust, distributed or parallel implementation and apply it to a production data set.

This workflow can require a tremendous amount of effort. In particular, moving to a distributed or parallel implementation of a large-scale learning algorithm is not easy, even using a dataflow platform such as Hadoop [44], Spark [46], DryadLinq [45], or Flink [6]. A programmer must work as a "human optimizer," making dozens of crucial design choices, most of which relate to questions such as what is an appropriate representation of data (especially intermediate results), which operations will be used to perform the computation (joins, maps, reduces, etc.), the order those operations are applied in, and also physical design choices such as which operations will have results cached in RAM or on disk for future use. In our experience, programmers have a very tough time planning such a computation. For just a few examples, we have seen cases where:

- In a text mining application written on top of Spark, a programmer inappropriately chooses to normalize the term-frequency vectors associated with each document in the corpus, storing each non-zero entry as a separate entry in an RDD. The result is that each step of a gradient descent algorithm requires an expensive join.
- In another Spark text mining application, a programmer does not cache an RDD that is used at each iteration of the algorithm, requiring it to be recomputed at each iteration.
- In an application mining spatial data, a programmer joins a large number of grid cells with information about the locations each grid contains, and then takes the top twenty cells, based on a metric that was known *before* the join. Had the top-k been run first, the join would have been costless.

These may be obviously poor programming choices, but most bugs are obvious in hindsight. Further, there is the problem that even once the programmer gets things right, changing data characteristics (or moving to an even larger data set) can render an optimal implementation suboptimal. Dependence of data manipulation code written in a non-declarative programming interface on the characteristics of the data it operates on has long been recognized as undesirable. In fact, the fragility of such non-declarative data processing codes was impetus behind the definition of the fully declarative relational calculus [19] and eventually the declarative subset of SQL—two inventions that began thirty years of dominance for the relational model.

A Declarative Language for Large-Scale Statistical Processing. While there have been notable efforts aimed at analyzing host languages to automatically build efficient dataflow programs [7, 28], this is provably an impossible task in the general case. A different option is to use a declarative language. However, SQL—the most widely-used declarative language—typically does not provide direct support for mathematical structures such as vectors and matrices. This is also true of extensions to SQL, such as Microsoft's U-SQL programming language for their Azure Data Lake services platform [1]. U-SQL is a general-purpose framework, and is not meant specifically for mathematical programming. Array databases

SIGMOD'17, May 14-19, 2017, Chicago, Illinois, USA

DOI: http://dx.doi.org/10.1145/3035918.3035937

that *do* support such structures (see [40] for a survey) generally lack support for declarative specification of complex, recursive computations.

In this paper, we describe the BUDS programming language, which is a statistically-oriented, declarative language. Our design for BUDS is motivated by the BUGS language for probabilistic programming [32]. BUDS can be used to declaratively specify a distributed Markov chain simulation whose state consists of millions or even trillions of values, and may require terabytes to store.[1]

We are interested in Markov chain simulation since it is the fundamental method for learning a statistical model in Bayesian ML, where it is known as Markov chain Monte Carlo, or MCMC [8]. While our emphasis is on Bayesian ML, BUDS can be used for many iterative learning tasks.

At first glance, BUDS looks something like a variant of R or MATLAB. In that sense, BUDS resembles SystemML [25] and Mahout Samsara [2] which are scalable, declarative, linear algebra systems. However, a key difference is that BUDS is not limited to vectors and matrices. BUDS allows declarative, distributed computations over sets, maps, arrays (including vectors and matrices), and compositions of these types. In practice, we find that this can significantly expand the set of computations that are easily specified using BUDS. For example, it is very difficult to code hidden Markov model learning over a large document corpus as a pure vector/matrix computation, as we detail in the Appendix of the paper.

Our Contributions. They are as follows:

1. The types supported in BUDS—vectors, arrays, etc.—are simply logical abstractions useful for programming, and do not correspond to the actual implementation. In fact, BUDS *automatically* chooses the physical realization of these abstractions in a distributed system. A large matrix may be physically stored as a set of column vectors; the optimal implementation is chosen automatically by the system.

2. Likewise, there are many available implementations of the abstract operations offered by BUDS (matrix multiplies, transposes, Hadamard products, etc.). These are tightly coupled with the physical representation—for example, a "pure matrix multiply" using the BLAS library [29] is only available for data stored as actual matrices, and not for a matrix stored as a set of vectors. In BUDS, these implementations are automatically co-optimized along with the representation.

   These two ideas: automatic optimization of data representation and co-optimization of the abstract operations offered by BUDS, allow for a programmer to write efficient BUDS programs with a minimum of effort.

3. Finally, we show that BUDS can be used to write four different Bayesian machine learning codes. We evaluate the efficacy of the BUDS optimizer on these codes.

2. BUDS OVERVIEW

In the next two sections of the paper, we describe the BUDS language in detail. BUDS' design follows a few core principles:

- The language should be declarative. It should have no control flow, which is difficult to optimize and automatically parallelize/distribute.
- The BUDS language itself should serve primarily as a way to describe how data are moved between user-defined functions.
- The language should have a small but useful set of composable data types: maps, lists, and arrays.
- Implicit parallelism should be provided in the form of comprehensions over those data types.
- As in other mathematical languages, operations over vectors and matrices should be part of the language.

[1]For the uninitiated, a Markov chain is a discrete-time stochastic process, where at time tick i the state of the system X_i randomly transitions to state X_{i+1}, in such a way that $\Pr[X_{i+1} = x_{i+1}|x_0, x_1, ..., x_i] = \Pr[X_{i+1} = x_{i+1}|x_i]$.

2.1 A Simple BUDS Program

Example Model. Our example centers on a simple Markov chain that simulates k people traveling around a set of n cities and visiting m restaurants. In this model, each individual travels independently from one city to the other. Once the person arrives in a city, she chooses one of the local restaurants and then moves to the next city, repeating the process indefinitely.

The simulation starts at iteration $i = 0$ by assigning each individual to a city by drawing a random value from a categorical distribution parameterized with the vector $\mathbf{s}$ of length n, which contains the probabilities of starting at any given city (Here, "$\sim$" should be read as "is sampled from"):

$$c_j^{(0)} \sim \text{Categorical}\,(\mathbf{s}) \text{ for each } j \in \{1, 2, \ldots, k\}$$

Then each individual chooses a restaurant in that city. Given the matrix $\mathbf{D}$, which contains n rows and m columns and denotes the probability of visiting each restaurant in a city (so that $D_{a,b} = 0$ if restaurant b cannot be found in city a), a restaurant is selected by drawing a value from a categorical distribution, parameterized with the row-vector corresponding to the assigned city $c_j^{(0)}$:

$$r_j^{(0)} \sim \text{Categorical}\left(\mathbf{D}_{c_j^{(0)}}\right)$$

For subsequent iterations $i = 1, 2, \ldots, ...$, the simulation chooses the next city using the n-by-n transition matrix $\mathbf{T}$ denoting the probability of moving from one city to another. The city is drawn randomly from a categorical distribution parameterized with the row-vector corresponding to the current city $c_j^{(i-1)}$:

$$c_j^{(i)} \sim \text{Categorical}\left(\mathbf{T}_{c_j^{(i-1)}}\right)$$

At each city, a restaurant is selected using a categorical distribution parameterized with the row-vector of $\mathbf{D}$ corresponding to the current city $c_j^{(i)}$, subject to the constraint that the probability of the previous restaurant $r_j^{(i-1)}$ is zero, to avoid visiting it again:

$$r_j^{(i)} \sim \text{Categorical}\left(\mathbf{D}_{c_j^{(i)}} \middle| p_{r_j^{(i-1)}} = 0\right)$$

Representing the Data in BUDS. To implement this model in BUDS, we begin by first specifying the base data set and variables. This is done in the `data` section of a BUDS code:

```
data {
  k: range(individuals);
  n: range(cities);
  m: range(restaurants);
  s: array[n] of real;
  D: array[n,m] of real;
  T: array[n,n] of real;
}
```

The `range` type is used to associate an index variable with a set of values. The index variable `n` is allowed to range over the integer

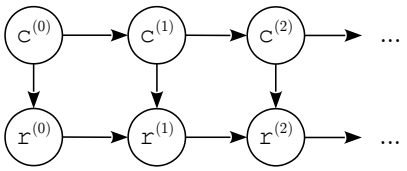

Figure 1: Graph of variable dependencies for the cities-restaurants model, iterations $i = 0, 1, 2$.

keys from the set $\{1, 2, \ldots, \texttt{cities}\}$. Users can then describe structures over such domains, such as the matrix `D`, defined as an `array` over the domains `cities` and `restaurants`.

Each value given as an argument to a `range` must be passed into the BUDS compiler at compile time (see Section 5.5); the set of `range` arguments are then used to cost candidate computational plans. Note that it is fine to omit the index variable, and, for example, simply declare `range(individuals);` without the `n`. The benefit of declaring an index variable, however, is that since the `range` argument must be supplied at compile time, its name is typically chosen so as to be externally meaningful. The index variable, however, is often chosen to match mathematical convention.

The random variables are then under the `var` section of the code:

```
var {
  c: array[k] of integer;
  r: array[k] of integer;
}
```

Describing Dependencies. BUDS differs from other mathematical languages such as MATLAB in that it is fundamentally declarative; to describe a computation, the programmer simply lists dependencies among variables. When a BUDS program is executed, variables are then updated recursively according to those dependencies. The set of dependencies can be represented as a directed graph in which vertices represent variable instantiations and edges represent relations of dependency. Figure 1 shows such a graph.

Since an iterative computation such as a Markov chain simulation must be initialized, BUDS provides syntax for describing initialization statements—that is, the variable assignments for the "zeroth" iteration of the computation. The BUDS description of the initialization for the variables $c^{(0)}$ and $r^{(0)}$ is:

```
init {
  for (j in 1:k) {
    c[j] <- categorical(s);
    r[j] <- categorical(D[c[j]]);
  }
}
```

The update assignments of $c^{(i)}$ and $r^{(i)}$ are then typically written at the end of the BUDS program (in our example code, the function `setEntry` is used to set the probability of the previously-visited restaurant to zero):

```
for (j in 1:k) {
 c[j] <- categorical(T[c[j]]);
 r[j] <- categorical(setEntry(D[c[j]], r[j], 0.0));
}
```

Note that this loop is an example of a comprehension [15]. Hence, it is a parallel construct, akin to MATLAB's `parfor` construct.

2.2 Another BUDS Example

We now give a complete BUDS implementation for an actual Bayesian ML algorithm. The Bayesian Lasso [37] is a Bayesian variant of the Lasso, using a regularizing prior on the regression coefficients. In our discussion, we assume that the base dataset is

```
data {
  n: range(responses);
  p: range(regressors);
  X: array[n,p] of real;
  y: array[n] of real;
  lam: real;
}

var {
  sig: real;
  b, t: array[p] of real;
  yy, Z: array[n] of real;
  A: array[p,p] of real;
}

init {
  sig <- invGamma(1, 1);
  t <- { invGauss(1, lam) | j in 1:p };
}

A  <- inv(X '* X + diag(t));
yy <- { y[i] - mean(y) | i in 1:n };
Z  <- yy - X * b;

b <- normal(A * (X '* yy), sig * A);
sig <- invGamma( ((n-1) + p) / 2,
      (Z '* Z + (b '* diag(t) * b)) / 2 );

for (j in 1:p) {
  t[j] <- invGauss(sqrt((lam * sig) / b[j]),
          lam);
}
```

Figure 2: BUDS specification for Bayesian Lasso learning.

comprised of a regressor matrix $\mathbf{X}$ with n rows and p columns, a response vector $\mathbf{y}$ of length n and the scalar, real-valued Lasso parameter $\lambda \geq 0$. The goal is to infer the vector of regression coefficients $\boldsymbol{\beta}$, the variance σ^2, and the vector of features $\boldsymbol{\tau}$.

The learning algorithm for the Bayesian Lasso is a special Markov chain simulation called a *Gibbs sampler*. Given the definitions $\tilde{\mathbf{y}} = \mathbf{y} - \mu_{\mathbf{y}}$, $\mathbf{D}_\tau = \text{diag}\,(\tau_1, \tau_2, \ldots, \tau_p)$, $\mathbf{Z} = (\tilde{\mathbf{y}} - \mathbf{X}\boldsymbol{\beta})$ and $\mathbf{A} = \left(\mathbf{X}^\top\mathbf{X} + \mathbf{D}_\tau\right)^{-1}$ as well as a suitable initialization for σ^2 and τ, the update statements for the corresponding distributions of the Gibbs sampler are:

$$\boldsymbol{\beta} \sim \text{MultivariateNormal}\left(\mathbf{A}\mathbf{X}^\top\tilde{\mathbf{y}}, \sigma^2\mathbf{A}\right)$$

$$\sigma^2 \sim \text{InverseGamma}\left(\frac{(n-1)+p}{2}, \frac{\mathbf{Z}^\top\mathbf{Z} + \left(\boldsymbol{\beta}^\top\mathbf{D}_\tau\boldsymbol{\beta}\right)}{2}\right)$$

$$\tau_j \sim \text{InverseGaussian}\left(\frac{\lambda\sigma^2}{\beta_j}, \lambda\right) \text{ for each } j \in \{1, 2, \ldots, p\}$$

This Markov chain can be specified in BUDS as in Figure 2. Note that this simple bit of code tracks the mathematics almost precisely. We begin with the definition of `A`, `yy` and `Z`, which correspond to $\mathbf{A}$, $\tilde{\mathbf{y}}$, and $\mathbf{Z}$ in the mathematical specification. We then give an initialization, as well as updates for `b`, `sig` and `t`, which correspond to β, σ^2, and τ in the math.

3. BUDS SYNTAX AND SEMANTICS

3.1 BUDS Data Types

A data definition binds a variable name to a data type that describes how the variable is structured. There are two kinds of data definitions: base data definitions, which are located in the `data`

section of the code and used to describe the domains and symbols of the base dataset, and *variable* definitions, which are located in the `var` section and describe the structure of random and temporary variables. The BUDS type system supports singleton types such as `integer`, `real`, `string`, the compound `array`, `list` and `map` types, and the special domain declaration and reference types `range` and `value`.

Let us consider the Bayesian Lasso. Simple elements such as the real-valued constant λ are easy to declare:

```
lam: real;
```

For the regressor matrix $\mathbf{X}$ and the response vector $\boldsymbol{y}$ to be declared, a description of the domains on which those compound structures are defined must be provided using the `range` type:

```
n: range(responses);
```

The above definition binds the symbol `n` to the domain of responses, indexed by the integer key attribute values $1, 2, \ldots,$ `n`. Note that `range` symbols can only be defined in the `data` section of the code, and whenever a `range` symbol is referenced in the context of a mathematical model expression, it denotes the maximum of those integer key values. Once `n` has been declared and associated with `responses`, both symbols can be used to describe structures with compound types, such as the response vector $\boldsymbol{y}$:

```
y: array[n] of real;
```

The above declaration defines `y` as an array of length `n` comprised of entries of `real` type. In general, `array` types are meant for describing dense structures, which means that, in a structure of the form `array[`$r_1, r_2, \ldots, r_k$`] of` T, there is a value of type T on each of the $r_1 \times r_2 \times \ldots \times r_k$ entries. In the case of the Bayesian Lasso, all the structures in the dataset and random variables happen to be dense, and therefore the `array` type is enough to describe said structures.

Other Compound Types. Other models require sparse, set-based structures. For instance, some text mining models have a base dataset consisting of a dictionary of words $w_1, w_2, \ldots, w_m$ and a corpus of documents $d_1, d_2, \ldots, d_n$ represented with the structure $\mathbf{z}$ where $\mathbf{z}_{i,j}$ is a positive integer denoting the number of times that word w_j appears in document d_i (the so-called "bag of words" model). Since the set of words contained in a given document usually corresponds to a small portion of the whole dictionary, a dense area `array` is not an adequate type for representing $\mathbf{z}$. BUDS provides the `map` type for such situations. The syntax for declaring a `map` is `map[`d`] of` T, where d is the name of the *key* domain. Although similar to `array` in most respects, `map` types can only be defined over a single key domain[2] and do not provide the guarantee that there always exists an entry of type T on each possible entry in the structure. Thus, this data set would be represented as:

```
data {
  n: range(documents);
  w: range(words);
  z: array[n] of map[words] of integer;
}
```

The other compound data type in BUDS is the `list` type, which is defined with the syntax `list[`d`] of` T, where d is the name of the indexing domain. The `list` type is used to represent array-like structures of variable length, which are useful for sequential objects, such as time series. In some text mining models, the position of a word within a document is important, so that each document in a data set is a variable-length sequence of words from a dictionary:

```
data {
  n: range(documents);
  w: range(words);
  p: range(wpos);
  z: array[n] of list[wpos] of value(words);
}
```

The above code defines each document as a list indexed over the domain `wpos`, with a maximum length of `p`. The `value` type is used to denote that each entry is a single value corresponding to the domain of `words`, as an integer in $\{1, 2, \ldots,$ `w`$\}$.

3.2 Expressing Computation

Assignments. A BUDS program is a set of dependencies; each node is a variable, and an edge is a direct dependency (expressed via the left arrow `<-` operator). A BUDS is executed as a series of *epochs*; each epoch corresponds to an update of each program variable according to the dependencies listed.

Recursion. Dependencies can be written in any order. With the exception of initialization statements, circular references between variables are permitted—in fact, such references are essential, as they describe recursion. Consider the following statements:

```
init {
  W <- f(c);
}
Z <- g(W);
W <- h(Z);
```

These specify a computation that begins with an initialization of `W` using a function that takes constant values as input. Thereafter, new instantiations of `Z` and `W` are iteratively generated using as parameters the previously-generated instantiations of said variables.

Variable Expressions. The expression on either side of an `<-` dependency may be the name of a variable or a reference to a particular entry in the variable structure using a temporary indexing variable declared within the context of a block. For example, we might separately applying an assignment expression on the rows of a matrix or the entries of a vector, as in the Lasso vector $\boldsymbol{\tau}$:

```
for (j in 1:p) {
 t[j] <- invGauss(sqrt((lam * sig) / b[j]), lam);
}
```

Here, the variable expressions `t[j]` and `b[j]` make use of the temporary index variable `j`, which is defined over the domain of those variables–that is, `regressors`. The above block does not denote a "loop" in the conventional sense; rather, it denotes independent, parallel assignments to separate entries of the vector `t`. Therefore, the use of temporary variables that change state on every iteration of the "loop" or any form of variable dependence among *different* cells of a compound variable inside the "loop" are invalid.

We return to the example from Section 2.1, where a user might be tempted to denote setting the probability of the previously-visited restaurant to zero as follows:

```
var {
  tempD: array[m] of real; ...
} ...
for (j in 1:k) {
  c[j] <- categorical(T[c[j]]);
  tempD <- D[c[j]];
  tempD[r[j]] <- 0.0;
  r[j] <- categorical(tempD);
}
```

[2]The reason why only one key domain is allowed is because BUDS provides specific syntax for accessing the set of keys present in a structure of type `map` using the domain name. Nonetheless, definitions of the form `map[`d_1`] of map[`d_2`] of` ... are acceptable.

The above code does not compile in BUDS since it is not possible to assign a variable to an expression more than once in an epoch. Note, however, that even after removing the line `tempD[r[j]] <- 0.0;`, this program would not compile, as `tempD` is being fully reset in the body of the loop, and hence iterations of the loop body cannot correctly be run in parallel.

However, the following *would* be valid:

```
var {
  tempD: array[k, m] of real;
  ...
}
...
for (j in 1:k) {
  c[j] <- categorical(T[c[j]]);
  tempD[j] <- setEntry(D[c[j]], r[j], 0.0);
  r[j] <- categorical(tempD[j]);
}
```

This is fine, as it is equivalent to the straight-line code:

```
  ...
  c[1] <- categorical(T[c[1]]);
  tempD[1] <- setEntry(D[c[1]], r[1], 0.0);
  r[1] <- categorical(tempD[1]);
  c[2] <- categorical(T[c[2]]);
  tempD[2] <- setEntry(D[c[2]], r[2], 0.0);
  r[2] <- categorical(tempD[2]);
  ...
```

Clearly, no variable is set twice.

Linear Algebra Operations. BUDS includes syntax for performing linear algebra operations over `array` types. The addition ("+") and subtraction ("−") operators can be applied on any two compound variables of the same type, and denote entry-wise arithmetic. The operators `.*` and `./` work similarly. Arithmetic operations between compound types and `integer` and `real` scalars are allowed, and denote the independent application of the operation on each element of the compound variables. For example, given the vectors $\mathbf{a}$ and $\mathbf{b}$ of length k and the scalar value x, the expression

$$x\mathbf{1}_k + (\mathbf{a} - \mathbf{b})$$

can be computed in BUDS as `x + (a - b)`, where `a` and `b` are both of type `array[k]`.

In addition to the operations outlined above, BUDS includes syntax for multiplication between matrices or vectors, possibly combined with transposition. For these operations, BUDS assumes that any `array[k]` value denotes a column vector of length k, and that an `array[m,n]` denotes a matrix with m rows and n columns. The multiplication operator $\star$ accepts two matrices with types `array[m,k]` and `array[k,n]` and returns a matrix of type `array[m,n]`. The transpose-multiply operators `'*` and `*'` have two applications: first, to allow for matrix multiplications of the form $\mathbf{A}^\top\mathbf{B}$ and $\mathbf{A}\mathbf{B}^\top$, respectively; and, second, as a requirement for vector products of the form $\mathbf{x}^\top\mathbf{y}$ (inner product) and $\mathbf{x}\mathbf{y}^\top$ (outer product), which is necessary as BUDS treats all `array[k]` types as column vectors. For example, the expression $\mathbf{A}\mathbf{X}^\top\tilde{\mathbf{y}}$ from the Bayesian Lasso can be represented in BUDS as:

```
A * (X '* yy)
```

where the `'*` operator produces an `array[p]` which, when multiplied with `A` on the left-hand side, results in an `array[p]`.

Aggregate Functions. Performing aggregation on compound structures in BUDS is achieved with the employment of the `sum`, `mean`, `var`, `stdev` and `count` functions. These functions take a structure defined over the domains $d_1, d_2, \ldots, d_n$ and return a structure defined over the domains $d_2, \ldots, d_n$. In the case where the structure is defined over a single domain, the result is a real-valued scalar, with the exception of `count` which returns an integer. Intuitively, BUDS aggregate functions can be understood as SQL aggregates with a Group-by clause that encompasses $d_1, d_2, \ldots, d_{n-1}$. Thus, for example, given the matrix `X` of type `array[m,n]`, the expression `sum(X)` returns an `array[n]` structure where the jth entry equals $\sum_i$ `X[i,j]`.

Consider, for example, the vector $\tilde{\mathbf{y}}$ from the Bayesian Lasso, which is computed by subtracting the mean $\mu_{\mathbf{Y}}$ from each element y_i. The BUDS comprehension assignment is straightforward:

```
yy <- { y[i] - mean(y) | i in 1:n };
```

3.3 Comprehensions and Parallelism

Any assignment under a `for` block can be represented using a comprehension syntax expression [15]. Comprehensions are a central feature of the BUDS language and are used to describe compound structures with a set of range definitions. In general, a comprehension is an expression of the form

$$\{e \mid r_1, r_2, \ldots, r_k\}$$

and is read as "the collection of all e where $r_1, r_2, \ldots, r_k$". For example, the `for` blocks shown above are equivalent to the following assignments using comprehensions:

```
init {
  t <- { invGauss(1, lam) | j in 1:p };
}
t <- { invGauss(sqrt((lam * sig) / b[j]), lam)
   | j in 1:p };
```

Note that `for` blocks and comprehensions can be nested arbitrarily. Thus, the block

```
for (i in 1:n) {
  for (k in 1:m) {
    W[i,k] <- f(Z[i,k]);
  }
}
```

can be represented with the comprehensions

```
W <- { { f(Z[i,k]) | k in 1:m } | i in 1:n };
```

which can be abbreviated as

```
W <- { f(Z[i,k]) | i in 1:n, k in 1:m };
```

In addition to range definitions of the form v_t `in` v_r, comprehensions also permit the use of boolean predicates for filtering elements, so that only the ones that satisfy said predicates are present in the structure. For example, given the variable `y` of type `array[n]`, it is possible to write a comprehension that defines a structure that only contains the positive values of `y` as follows:

```
v <- { y[i] | i in 1:n, y[i] > 0 };
```

Since the structure defined in the above comprehension will possibly contain less than `n` entries, it cannot be treated as a dense structure anymore, and therefore the application of a boolean predicate on any `array` structure produces a structure of type `map` defined over the same domain. In the case of `map` and `list` types, no such type demotion is applied.

Comprehensions provide many benefits. They are an elegant construct for describing an entire model as a set of simple assignments. Comprehensions are also central to the BUDS model of parallelism. The right-hand side of the comprehension expression defines the level of granularity of the computation, so that each instance of the expression on the left-hand side can be computed independently and then "collected" together as separate component of a larger structure.

The fact that comprehensions are executed in parallel and do not allow for dependencies between "iterations" can be a bit difficult for a programmer. For example, consider a Bayesian Hidden Markov Model for text (see Figure 5 in the Appendix). Since the algorithm for learning such a model requires associating hidden states with each word in the text, and those hidden states have statistical dependencies on their neighbors, it is not possible to perform this assignment massively in parallel. In this case, our solution was to implement a user-defined function that operates on each document as a single, indivisible unit.

4. EXECUTION OVERVIEW

4.1 Choosing a Prototype Platform

Our goal is to design the BUDS compiler to sit on top of an existing execution platform—BUDS is a front-end programming interface, and not a replacement for existing dataflow or relational database engines. Still, we had to choose a prototype backend. It was clear early on that one of the key components of a BUDS implementation is a cost-based optimizer to determine the quality of various backend implementations. Since available, open-source dataflow platforms lack a full-fledged, cost-based optimizer, we chose a relational database system.

In particular, we settled in SimSQL as the backend [17] for two main reasons. First is SimSQL's support for a very flexible type of user-defined function called a *VG function*, which allows relatively complex constructions taking multiple tables as input, and producing multiple tables as output. Second is SimSQL's support for recursively-defined tables, which make it a natural target for fixpoint computations.

After choosing SimSQL, we asked, should we translate BUDS directly into relational algebra (which will then be optimized and executed by the relational database) or should the target be SQL? After much thought, we decided on SQL as an intermediate. SQL-to-relational algebra translation is a difficult task in itself, and the translation involves many steps that are likely to be repeated in any BUDS direct-to-relational translation, such as unnesting via magic sets re-writing [9].

4.2 Translator Overview

Given these considerations, we designed the BUDS translator so that it takes as input the stochastic model together with statistics describing the size of each of the domains referred by variables in the model, then executes a sequence of steps that produces

(1) A schema for storing the base dataset in a set of relational database tables, presented to the user as a series of SQL `CREATE TABLE` statements.

(2) A set of queries for initializing and generating samples for the variables in the model. For each of the variables, a SQL statement describing a derived relation is returned.

SQL is at a lower level of abstraction than BUDS, in that SQL typically supports tables of records, and the logical types supported by BUDS (maps, lists, arrays) need to be implemented in terms of tables. Thus, there are many possible SQL implementations for a BUDS program. The BUDS optimizer searches among those, invoking the optimizer for each. The SQL optimizer is then used as a black box to search among different relational algebra implementations for the resulting SQL program. The two key technical difficulties of the translation process are:

(1) The types supported in a BUDS do not correspond to the physical implementation of the program. Thus, the BUDS compiler must automatically choose the physical realization of these abstractions. For example, a large matrix may be physically stored as a set of column vectors; the choice of an optimal implementation should be automatic.

(2) Likewise, there are many available implementations of the abstract operations offered by BUDS, which depend upon the physical implementation—for example, a matrix multiply using the BLAS library [29] is available for data stored as a matrix, and not for a matrix stored as a set of vectors, where the multiplication would be implemented as a set of inner products over the output of a join:

```
SELECT A.row_id, B.col_ID,
  inner_product (A.vec, B.vec)
FROM A, B
```

Our central idea is to explore the space of alternative implementations using an A*-style search algorithm [38]. That is, we first translate the source specification into a target code whose semantics are equivalent, and then we employ a series of transformations that move from one target implementation to another. For each solution that the search algorithm evaluates, the translator generates SQL code and passes it to the DBMS's query compiler and query optimizer. The query optimizer returns the execution cost of the solution back to the translator and the process is repeated until a solution of minimal cost is obtained and returned to the user. This idea is reminiscent of the tactic pioneered by the developers of Microsoft SQLServer's AutoTune wizard [18].

4.3 The Target Platform: SimSQL

We chose SimSQL due to its support for user-defined functions and recursion. In the remainder of this section, we give an overview of SimSQL's support for both. One thing missing from SimSQL is native support for vectors and matrices, which is crucial to producing efficient statistical codes. Thus, we also briefly describe our SimSQL extensions for vectors and matrices.

We illustrate SimSQL VG functions as well as recursion by returning to the simulation from Section 2. Let us assume that we have the following table, which encodes the vector **s** containing the starting probabilities for each city:

```
STARTPROBS(DIM, VAL)
```

`DIM` tells us the position in the vector, and `VAL` is the value in that position. We also have a table listing all of the people:

```
INDIVIDUALS(PID)
```

The first city is chosen using the Categorical VG function as follows:

```
CREATE TABLE CITY[0] (PID, CID) AS
  FOR EACH i IN INDIVIDUALS
    WITH Res AS Categorical (
      SELECT * FROM STARTPROBS)
    SELECT i.PID, r.CID
    FROM Res AS r
```

Briefly, what this code does is to consider every tuple `i` in the `INDIVIDUALS` table. For each individual, the `Categorical` VG function is parameterized with all of the city probabilities from `STARTPROBS`, which it uses to select a city at random. This result is stored in the table `Res`. For a given `i`, the final `SELECT` query then creates a tuple which is added to the `CITY` table. In the general case, more than one tuple can be added, but here it is exactly one. The tuples produced by all of the executions of the `SELECT` statement (one for each individual) are `UNION`ed together to create the `CITY[0]` relation.

Ignoring (for brevity) how restaurants are selected, we can then move all of the people to the next city by conditioning on the cur-

rent city. We assume a CITYPROBS(FROM_CID, TO_CID, VAL) table that encodes the **T** matrix and gives us the probability of transitioning between cities. Then we have:

```
CREATE TABLE CITY[i] (PID, CID) AS
  FOR EACH i IN INDIVIDUALS
    WITH Res AS Categorical (
      SELECT cp.TO_CID, cp.VAL
      FROM CITYPROBS AS cp, CITY[i - 1] AS c
      WHERE cp.FROM_CID = c.CID AND c.PID = i.PID)
    SELECT i.PID, r.RID
    FROM Res AS r
```

The probabilities that govern the transition to the next city is chosen by looking at the last city that person i was located in (that is, join CITY[i - 1] with the tuple i on c.PID = i.PID).

4.4 Vectors and Matrices

To achieve efficient execution of the code from the BUDS-to-SQL translation task, it is important that vectors and matrices are supported as native attribute types.

Consider the problem of computing a Gram matrix from a list of vectors storing bag-of-words encodings of a set of documents (where each distinct word present in the document is mapped to an entry in the vector). If $\mathbf{x}_i$ stores the ith document as a row vector, the gram matrix is computed as $\sum_i \mathbf{x}_i^T \cdot \mathbf{x}$. In "vanilla" SQL, with the table DOCUMENTS(DOC_ID, DIM_ID, VAL) storing the list of vectors, this would be expressed in SQL as:

```
SELECT SUM (x1.VAL * x2.VAL), x1.DIM_ID, x2.DIM_ID
FROM DOCUMENTS x1, DOCUMENTS x2
WHERE x1.DOC_ID = x2.DOC_ID
GROUP BY x1.DOC_ID
```

This is expensive, since it requires a join of (potentially) a very large table, followed by an aggregation. Further, if each document averages m distinct words—m can easily be on the order of 1000—aggregating n documents requires processing $n \times m^2$ tuples output from the join. This can be debilitating.

In our extension to SQL (which we call *eSQL*) we can instead store such vectors in the table DOCUMENTS(DOC_ID, WORDS) where WORDS is a vector. The Gram matrix code is simply:

```
SELECT SUM (OUTER (WORDS, WORDS))
FROM DOCUMENTS
```

This is much more efficient, requiring a simple scan of the DOCUMENTS table. As an alternative, DOCUMENTS could have a single attribute ALLDOCS, which is a matrix type. In this case, the gram matrix computation is simply:

```
SELECT MATRIX_MULTIPLY (TRANSPOSE(ALLDOCS), ALLDOCS)
FROM DOCUMENTS
```

In this case, the matrix multiply is implemented using BLAS [29].

The extended SQL also contains facilities for constructing vectors and matrices. For example, a simple query to construct a matrix from CITYPROBS(FROM_CID, TO_CID, VAL) is:

```
CREATE VIEW CITYPROBS_MATRIX (VAL) AS
SELECT ROWMATRIX (MROW)
FROM (SELECT LABEL(VECTORIZE(LABEL(TO_CID, VAL)),
                   FROM_CID) AS MROW
      FROM CITYPROBS
      GROUP BY FROM_CID)
```

Here, the inner query creates a vector for each FROM_CID using the VECTORIZE aggregate function. These vectors are then labeled with their row identifier (the FROM_CID) and aggregated into a single tuple with a matrix attribute using the ROWMATRIX aggregate function. Vectors and matrices can be deconstructed as well:

```
SELECT c.CNT AS ROW, GET_ROW (c.CNT, cm.VAL)
FROM COUNTS AS c, CITYPROBS_MATRIX AS cm
```

Here, COUNTS is a system table, containing tuples with values 1, 2, 3, etc.

5. TRANSLATING BUDS MODELS

In general, the compilation process begins by parsing the input and performing semantic checks. Next, a data dependency graph is created. It is then analyzed to check for correct initializations and lack of cyclic dependencies. At this point, the translation can begin.

5.1 Moving Among Data Representations

Programmers in BUDS choose from data structures such as arrays, matrices, maps, etc. One of the most important tasks in the translation process is choosing how these data structures can be represented in the underlying relational or linear model. For example, the matrix D from BUDS in our city-and-restaurants model can be represented in eSQL using four possible different schemas:

1. A table with $n \times m$ records, each containing a double attribute with the value of the cell $D_{i,j}$, an integer attribute with the key value for city i, and another integer attribute with the key value for restaurant j; or,
2. A table with n records, each containing a vector attribute of size m with the values of the row vector $\mathbf{D}_i$, and an integer attribute with the key value for city i; or,
3. A table with m records, each containing a vector attribute of size n with the values of the column vector $\left(\mathbf{D}^\top\right)_j$, and an integer attribute with the key value for restaurant j; or,
4. A table with a single record containing a matrix attribute of size m, n with the entire contents of $\mathbf{D}$.

It is incumbent upon the translation framework to automatically choose a suitable representation—one that can be implemented efficiently by the underlying platform.

The most fundamental data structure used to move among representations to choose the optimal one is the *type graph*, which is a directed graph where an edge between two types indicates that it is possible to generate code that directly moves between them. Since certain types are incompatible with one another, this graph is almost assuredly going to be disconnected for most translation tasks. Part of the BUDS-to-SimSQL typeGraph relation is depicted in Figure 3. This shows four possible representations for a matrix (or two-dimensional array) in BUDS.

Each edge in this graph has an associated eSQL code template that can be used to generate code for moving between concrete types. We describe how the type graph is used to move among various eSQL representations for a BUDS type logically, using Datalog (in our actual BUDS implementation, we use Prolog).

The codes associated with edges in the type graph are represented in Datalog via the relation:

```
xformImp(InType, OutType, InName, OutName, Str).
```

An entry in this relation means that it is possible to transform the variable InName of type InType into the variable OutName of type OutType using the code contained in the string Str. For example, consider the edge from v_1 to v_2 in Figure 3, which corresponds to moving from a purely relational matrix representation, to a set of vectors. We have the corresponding Datalog rule (here, we use the convention that identifiers beginning with lower-case letters are literals, and those with upper-case are variables; + refers to the string concatenation operation):

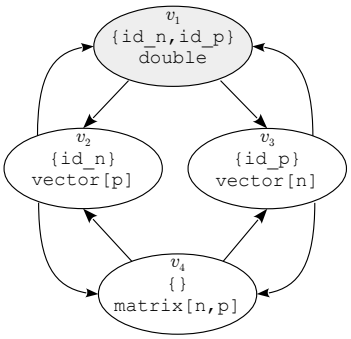

Figure 3: Type graph over possible representations of the matrix variable X.

```
xformImp(v1, v2, InName, OutName, Str) :- Str =
  "CREATE VIEW " + OutName +
  "AS SELECT inp.id_n AS id_n, VECTORIZE(
        LABEL(inp.val, inp.id_p)) AS val
  FROM " + InName + " AS inp
  GROUP BY in.id_n;".
```

This rule describes how the actual code string `Str` is constructed by inserting the `InName` and `OutName` into the eSQL code. As described previously, the special-purpose SimSQL aggregate function `VECTORIZE` takes a set of labeled numeric attribute values and creates a single `vector` type, indexed using the each value's label as assigned using the `LABEL` function.

We can go the other direction as well.

```
xformImp(v2, v1, InName, OutName, Str) :- Str =
  "CREATE VIEW " + OutName + " AS
  SELECT inp.id_n AS id_n, r.id_p AS id_p,
         GETSCALAR (inp.val, r.id_p) AS val
  FROM " + InName + " AS inp, responses AS r;".
```

Here, the table `responses` contains p records with the key values of `id_p`, which are used by the `GETSCALAR` function to obtain individual entries from the `vector` value `v2.val`.

We not only want to be able to traverse one edge in the type graph to change representations, but we want to be able to take multiple hops. If a path from `InT` to `OutT` exists, we can generate code for it using the following rule:

```
xformImp(InT, OutT, InName, OutName, Str) :-
  Str = xformImp(InT, IntmedT, inName, temp) +
        xformImp(IntmedT, OutT, temp, outName),
  temp = InName + OutName + OutT.
```

Here, `inName + outName + OutT` is used to create a unique identifier for the table holding intermediate results.

5.2 Searching for Implementations

The translator must not only be able to search among data representations: it must search among implementations of BUDS operators that run over those representations, and apply those implementations. The reason for this dual search (representations and implementations) is that representations and their associated implementations may be good sometimes, but not all of the time. For example, a direct eSQL matrix multiply that is implemented as a call to BLAS is likely optimal for large matrices that are small enough to fit in memory. But it will fail for very large matrices.

We do this using an abstraction called an *expression graph*. This is an expanded version of the data dependency graph, where, in addition to all named variables, dependencies among all temporary variables that exist only as the output of particular operations appear as well. Vertices without any input edges, which we will call "leaf" vertices (although the structure might not be a tree) represent data–in the case of BUDS, they are variables from the `data` and `var` sections of the program. Non-leaf vertices represent operations. Edges represent flows among operations. For example, in the case of BUDS, consider the expression graph for the following portion of the Bayesian Lasso:

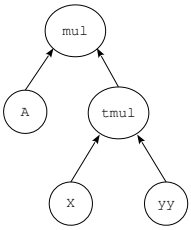

Figure 4: Expression graph for part of the Bayesian Lasso.

```
A * (X '* yy)
```

The corresponding expression graph is shown in Figure 4.

To describe how this graph is used to power the search for implementations, we will ignore how user-defined function calls are handled; these have an arbitrary number of parameters and hence make the presentation a bit more muddled. Excluding these, the graph can be represented as three Datalog relations:

```
leafNode (VarName, Type).
unNode (OpName, InName, OutName).
binNode (OpName, LName, RName, OutName).
```

These relations have the obvious meaning. `leafNode` lists the leaf nodes in the graph, `unNode` lists the unary, internal nodes in the graph, and `binNode` lists the binary nodes. For the later two operations, `OpName` is the name of the operation that needs to be run (such as `tmul`), and the various `Name` parameters give the names of the variables in the graph.

For each particular operation, we have one or more implementations, encoded in the following relations:'

```
binImp (OpName, TypeL, TypeR, OutType, LName,
  RName, OutName, Str).
unImp (OpName, TypeIn, OutType, InName,
  OutName, Str).
```

These relations are analogous to the `xformImp` relation, except that they provide code for implementing operations rather than moving between types. Note that, since operations are polymorphic (for example, matrix multiplication can be defined over many input types), there may be many entries in the relation for each operation name—implementations may operate on different input types and produce different output types. For example, we may have an implementation of `tmul` that accepts two sets of vectors, and another that accepts two matrices.

The various `Imp` relations store for us all available implementations, but it is also necessary to store instantiated implementations associated with our particular translation problem. The relation `imp (Type, Name, Str)` will accomplish this, storing in the string `Str` for all available implementations of the node with name `Name`, where the result takes the type `Type`. The first two rules simply abstract away whether the operation is unary or binary, and allow us to simply obtain its implementation code:

```
imp (Type, Name, Str) :-
  binImp (OpName, TypeL, TypeR, Type, LName,
  RName, Name, Str).
imp (Type, Name, Str) :-
  unImp (OpName, TypeIn, Type, InName,
  Name, Str).
```

One possible implementation is to do nothing, in case there is a leaf

node:

```
imp (Type, Name, Str) :- leafNode (Name, Type, "").
```

And we can possibly run code to perform a type transformation:

```
imp (Type, Name, Str) :-
  imp (InT, Name, InStr), InName = Name + Type,
  xformImp(InT, T, InName, Name, TransStr),
  Str = InStr + "ALTER VIEW " + Name " RENAME TO "
        + InName ";" + TransStr.
```

We can then use these implementations in conjunction with specific definitions of `binImp` and `unImp` to build up all possible code strings for a given query graph. For example, consider the operation `tmul` where a matrix is transposed and multiplied with a vector. If the input matrix is represented as a relation of tuples containing vectors (type v_2) and the input vector is represented as a set of tuples containing `double` values (imagine this is type v_{12}), then we might have the following rule in `binImp`:

```
binImp (tmul, v2, v12, v2, LVar, RVar, OutVar, Str)
:- imp (v2, LVar, LStr), imp (v12, RVar, RStr),
  binNode (tmul, LVar, RVar, OutVar),
  Str = LStr + RStr + "CREATE VIEW " + outVar +
  "AS SELECT SUM(X.val * yy.val) AS val
    FROM " + lVar + " AS X, " + rVar + " AS yy
    WHERE X.id_n = yy.id_n;".
```

This simply performs a vector-scalar multiply on each entry, then sums the result to obtain the output, which is a single tuple containing a vector. Or, a simple matrix transpose directly on a set of matrices will use the rule:

```
unImp (trans, v3, v3, InVar, OutVar, Str) :-
  imp (v3, InVar, InStr),
  unNode (trans, InVar, OutVar),
  Str = InStr + "CREATE VIEW " + outVar +
  "AS SELECT TRANSPOSE(X.val) AS val
    FROM X AS " + inVar + ";".
```

Note that the above code creates a unary operator (such as a matrix transpose) as two separate queries; the SQL query that creates the matrix to be transposed, and the query to transpose the matrix. This may seem costly. However, SimSQL aggressively pipelines such queries, meaning that the set of matrices produced by the first query would be pipelined directly into the transpose operation, expectedly at little cost.

5.3 Encoding Domain Specific Optimizations

Encoding domain-specific optimizations in this framework is easy. For example, consider the case of a Gram matrix computation over a matrix: $\mathbf{X}\mathbf{X}^T$, which corresponds to a unary `tmul` operation. Many different implementations of this computation are available. While the direct computation $\mathbf{X}\mathbf{X}^T$ is typically desirable because SimSQL will use BLAS to implement it directly, sometimes this implementation is not possible because $\mathbf{X}$ is too large to fit into RAM.

As an alternative, $\mathbf{X}$ can be represented as a set of vectors, and the Gram matrix computed as $\sum_i \mathbf{x}_i^T \cdot \mathbf{x}$. This special transformation can be represented as a Datalog rule:

```
unImp (tmul, v4, v4, InVar, OutVar, Str) :-
  imp (v4, InVar, InStr),
  unNode (tmul, InVar, OutVar),
  IntName = InVar + OutVar,
  xformImp(v4, v2, InName, IntName, TransStr),
  Str = InStr + TransStr +
    "SELECT SUM (OUTER (val, val))
     FROM " + IntName + ";".
```

Essentially, the above rule says that we can obtain a pure-matrix to pure-matrix `tmul` operation by first transforming the pure matrix to a set of vectors, and then performing a `SUM` of outer products.

5.4 Additional Details

The previous subsection described how all possible implementations for a given source program can be produced. As described previously, the tactic that we employ is to generate each of those possible implementations, send each to SimSQL's optimizer to cost them, and then we actually run the most inexpensive implementation. We rely on the optimizer—which has information about array sizes, if they are available—to detect infeasible implementations, such as the materialization of a huge matrix that cannot fit into RAM.

Another issue is that, in practice, there may be many thousands of valid implementations, and it is not practical to generate and cost each of them. We have found that in fact, a purely greedy algorithm works very well. We maintain a single best implementation, and then use the rules described in the previous subsections to generate all possible implementations reachable by changing one implementation or data representation. Each of those is costed, and the lowest-cost alternative is chosen as the new implementation. This is repeated until a fixpoint.

BUDS programs require statistics to optimize a computation correctly. These are supplied to BUDS by passing BUDS the various `range` values declared in the program. These are then used to compute the counts and the distinct values as the statistics for each of the relational base tables that are sent to the SimSQL optimizer for evaluation. For example:

```
> load models/lasso.txt
Model 'Lasso' successfully loaded.
> size lasso.regressors 1000
Domain lasso.regressors, size 1000
> size lasso.responses 1000000
Domain lasso.responses, size 1000000
> compile lasso
```

The output of this execution is: (1) a file of `CREATE TABLE` statements for the base tables that the user is expected to load data into, (2) a set of materialized view statements, and (3) a set of stochastic `CREATE TABLE` statements to power the simulation.

6. EXPERIMENTAL EVALUATION

In this section, we describe an experimental evaluation of the performance of the BUDS language and compiler for a set of representative Bayesian ML problems. The key task will be to discover BUDS-encoded programs comparing performance-wise to programs written directly on top of SimSQL in eSQL. Does the high-level interface provided by BUDS generate programs that are as performant as hand-coded programs, written by an expert?

6.1 Implementation Overview

The BUDS compiler/optimizer prototype is currently implemented around 17,000 lines of Java and Prolog. It accepts as input a BUDS program, and then produces as output a SimSQL eSQL program. Our current implementation has 16 different eSQL representations of BUDS data types available (examples include: a matrix stored purely as tuples, a matrix stored as a set of vectors, a map stored purely as tuples, etc.), and 57 different eSQL implementations of built-in BUDS operations (examples include: a pure matrix inverse, scalar-vector-as-tuple multiplication, etc.), as well as 26 different distribution functions (vector-based Dirichlet distribution, tuple-based Categorical distribution, etc.).

6.2 Experimental Overview

Our experimental task is implementing four different Bayesian ML models, which will be described subsequently. We evaluate five different eSQL implementations of each of the four models.

Model	Mode	Search Space Size	Opt. Time
BL	full	78 codes	00:08:57
BL	greedy	4 codes	00:00:33
GMM	full	1214 codes	08:54:27
GMM	greedy	48 codes	00:20:34
LDA	full	224 codes	00:15:22
LDA	greedy	24 codes	00:01:37
HMM	full	32 codes	00:01:51
HMM	greedy	8 codes	00:00:27

Table 1: Summary of optimization complexity for the four models. For each model, and for each optimizer mode (full search or greedy search) we give the number of SimSQL codes generated by the BUDS translator, as well as the time taken to generate and cost all of those codes (HH:MM:SS).

(1) Hand-coded, naive eSQL, no vectors/matrices. Since BUDS is translated into SimSQL's eSQL, we begin by hand-coding each ML algorithm directly in SimSQL's eSQL as a baseline. "Naive" here means that we do not anything special that might speed execution, such as grouping data items together and writing special-purpose VG functions that process the data as a group (see below).

(2) Hand-coded, tuned eSQL, no vectors/matrices. This is a carefully-tuned implementation, subject to the constraint that the implementations do not directly use vectors and matrices to store and manipulate data. Instead, a purely tuple-based encoding is used (for example, a 100 by 100 matrix is always stored as 10,000 tuples in the hand-coded implementations).

(3) Hand-coded, tuned eSQL, with vectors/matrices. In this implementation, we make full use of eSQL's vector and matrix support, and craft a carefully-tuned, hand written version. Given our understanding of eSQL vector/matrix support, this is likely to be the fastest implementation.

(4) BUDS-generated, non-optimized eSQL, with vectors/matrices. Here we do not iterate through the implementations generated as described in the previous section, costing each. Rather, we simply use the first implementation generated.

(5) BUDS-generated, optimized eSQL, with vectors/matrices. Here, full optimization is used to search the space of types and implementations.

Experimental Platform. All running times reported were obtained by running the SimSQL SQL codes using SimSQL running on a cluster of five Amazon EC2 m2.4xlarge machines.

6.3 Models Tested

Our experiments focus on the implementation of Markov chain simulations for learning the following four Bayesian models:

Bayesian Lasso. The BL has already been described previously, and the BUDS code for the model was given earlier as well. To evaluate the various implementations, we created a synthetic data set consisting of 500,000 data points distributed across the five machines, having 1,000 regressor dimensions each. Since there is not an obvious way to optimize the BL implementation without vectors and matrices, this option was not tested.

Bayesian Gaussian Mixture Model. The GMM assumes that the data set was generated by a mixture of Gaussians (multi-dimensional normal distributions) and the task is to recover the various components from the data. We use ten clusters to process a ten-dimensional data set, using a full covariance matrix. Fifty million synthetically-generated data points are distributed across the five machines. Two

Model	Implementation	Code Lines	Run Time
BL	Naive SQL	100	00:07:28 (02:38:46)
BL	Ve/Ma SQL	104	00:04:04 (00:04:44)
BL	BUDS no-opt	30	00:13:41 (02:41:25)
BL	BUDS opt	30	00:05:58 (00:20:22)
GMM	Naive SQL	197	00:21:16 (00:11:22)
GMM	Block SQL	161	00:06:39 (00:13:08)
GMM	Ve/Ma SQL	111	00:09:15 (00:08:09)
GMM	BUDS no-opt	39	00:20:27 (00:14:02)
GMM	BUDS opt	39	00:11:12 (00:11:35)
LDA	Naive SQL	126	14:32:04 (08:45:14)
LDA	Doc-based SQL	117	04:12:37 (03:59:26)
LDA	Ve/Ma SQL	101	00:29:28 (00:01:13)
LDA	BUDS no-opt	31	13:54:32 (17:13:55)
LDA	BUDS opt	31	00:37:42 (00:19:47)
HMM	Naive SQL	131	08:17:07 (10:51:32)
HMM	Doc-based SQL	123	03:36:47 (00:17:51)
HMM	Ve/Ma SQL	122	00:28:17 (00:26:28)
HMM	BUDS no-opt	33	00:45:32 (01:08:28)
HMM	BUDS opt	33	00:30:08 (00:05:52)

Table 2: Performance and code size of the various implementations. All times are given as HH:MM:SS per iteration. The value in parentheses is the time for the initialization iteration.

no-vector/matrix SQL implementations are tested: a naive implementation, and a second, highly-optimized implementation that updates the membership of a large block of data points using a single user-defined function call (see [16]).

Latent Dirichlet Allocation. LDA is a very standard topic model for unsupervised learning over text. We use a non-collapsed Gibbs sampler to learn this model (again, see [16] for details). For brevity, we do not give the BUDS code. 12.5 million documents are distributed across the five machines. A dictionary size of ten thousand words is used to learn 100 topics. Again, we have two SQL, no vector/matrix implementations: a naive implementation, and a second, optimized implementation that has a special user-defined function that determines the word-topic membership for each word in the document using a single function invocation. The BUDS implementation assumes a similar user-defined function.

Hidden Markov Model. Here we learn a Bayesian HMM over text. The data used are identical to the data used for LDA, but in the case of a HMM, 20 latent states are used. As in LDA, we have two SQL, no vector/matrix implementations: a naive implementation, and a second, optimized implementation that has a special user-defined function that determines the assignment of states to words all-at-once for a single document. In the case of BUDS, a similar user-defined function is used.

6.4 Results

For each of the four models, we give the search space size and optimization time in Table 1. The search space size is the number of distinct expression graphs that can be generated using the current set of BUDS data representations and implementations. We also have the optimization time (that is, the time to search the space) for two different search strategies: full and greedy. The full strategy exhaustively enumerates all alternatives. The greedy strategy is as described in the previous section of the paper.

In Table 2 we give the per-iteration running time and code size for each of the different implementations tested. All of the implementations are equivalent in the sense that they run the same algo-

1 (2.5M)	5 (12.5M)	10 (25M)	20 (50M)
00:45:13	00:37:42	00:46:08	00:45:05

Table 3: Running time per iteration when increasing cluster size, keeping documents per machine constant. Column header format is num. machines (total data set size).

0.625M	1.25M	2.5M	5M
00:16:47	00:29:41	00:37:42	01:25:13

Table 4: Running time per iteration when keeping cluster size constant, increasing corpus size. The column header is the number of documents assigned per machine, for each of the five machines in the cluster.

rithm; the only difference is in the details of the implementation and hence in the running time.

Scalability. In addition, we ran a set of experiments aimed at testing the scalability of the BUDS LDA implementation. We have two experiments. In the first (Table 3) we try four different cluster sizes: one machine, five machines, ten machines, and twenty machines, keeping the number of documents per machine constant at 2.5 million. If BUDS LDA scaled perfectly, the running time would be constant across these four tests. In the second (Table 4), we ran our five-machine experiment with four different corpus sizes: 625 thousand documents per machine, 1.25 million per machine, 2.5 million per machine, and 5 million per machine (25 million documents total). If BUDS LDA scaled perfectly with increasing data set size, we would expect that the time per iteration would increase linearly with corpus size.

6.5 Discussion

It is not surprising that ultimately, one of the hand-coded versions had superior performance in every case. Sometimes, the fastest hand-coded version was a blocked or document-based implementation, and sometimes the vector/matrix implementation was best. The ratio of the time obtained by the optimized BUDS code compared to the best hand-coded implementation for non-initialization iterations was 1.46, 1.68, 1.03, and 1.065 for BL, GMM, LDA, and HMM, respectively. We argue that these are not overly worrisome slowdowns, given the much greater simplicity of the BUDS code.

There was more variance in the time spent in the initialization iteration for each implementation, but the initialization is run only one time, whereas anywhere from 10 to 500 non-initialization iterations are needed to obtain a fixpoint. Thus, initialization times seem less important.

It is also interesting to note that in nearly every case, the vector/matrix extensions resulted in faster, hand-coded codes than the codes without vector/matrix extensions. The one exception to this is the GMM, where the block implementation (by definition) performs a significant fraction of the computation within a VG function, in C++, where it uses vectors and matrices internally. In this implementation, a special version of the `categoricalGMM` function is used that is parameterized only once for a block of data points, meaning that the cost of the parameterization is amortized across many data points, resulting in a highly efficient implementation. This mitigates the benefit of the vector-matrix extensions. However, had we implemented a block-based vector/matrix version of the GMM, undoubtedly it would have been faster. We believe that this shows the importance of native vector/matrix support in any system performing statistical computing.

The results also show the importance of the optimization of representation and implementation. In every case, the BUDS optimization process is able to arrive at a data representation and implementation that is far superior to the first solution obtained, without optimization. In the case of the HMM, the non-optimized version is about 1.5 times slower than the optimized. But in the case of LDA, it is more than 25 times slower.

One of the interesting findings is that the greedy strategy does a very good job of choosing a high-quality plan from among all of the alternatives, when pure greedy can fail spectacularly in classical query optimization. One explanation is that the problem of choosing from among data representations and physical operation implementations during the implementation of a language such as BUDS may actually be *easier* than classical query optimization—assuming that a classical query optimizer is available to optimize the SQL implementation chosen by BUDS. There is no structural plan search during BUDS optimization as there is when choosing a join ordering—the dependency graph among various data objects is fixed, and the problem is labeling the objects and operations with implementations. At the very least, there are fewer alternatives to consider when optimizing a BUDS program than there are for a relational algebra plan for a problem of similar size.

Final Remarks. Generally, BUDS did an exceptionally good job of discovering a reasonable implementation for each algorithm, considering the difficulty of the optimization problem. Consider the LDA learning algorithm. The core of the LDA simulation is a stochastic assignment of a topic label to every word in the corpus. Then, all of those topic labels need to be aggregated. What is the best way to do this? Should the words be aggregated into vectors according to the topics (one entry per word for each topic vector) and then summed? This would make sense, as subsequently, the vectors need to be summed to obtain a vector of word sums per topic. Or, perhaps it makes sense for those to be left as (word, topic) pairs and aggregated directly and *then* transformed into vectors? These are all difficult decisions to make, where the wrong choice can result in an implementation that is an order of magnitude slow. Such choices are made more complicated by the fact that attributes such as the dimensionality of the data (number of words in the dictionary and topics being learned, in the case of LDA) all have an effect on what the best implementation is. BUDS takes these difficult choices away from the programmer. We believe these results have presented at least some evidence here that BUDS can automatically do a reasonable job of choosing an efficient implementation.

7. RELATED WORK

There exist a number of APIs for building machine learning computations, including TensorFlow [5], Theano [10], and DistBelief [23]. These three systems focus on deep learning as the primary application. In addition, SystemML [25] and Mahout Samsara [2] are scalable linear algebra systems, that focus on machine learning algorithms requiring the use of very large, distributed matrices. The latter two systems are closest in spirit to BUDS. However, BUDS allows declarative, distributed computations over a richer set of types: sets, maps, arrays, and compositions of these. We believe that this significantly expands the set of computations that are easily specified using the system. This issue is considered in detail in the Appendix of the paper.

The development of the BUDS language is related to existing research in probabilistic programming languages. "Probabilistic programming" languages, broadly defined, are any languages that naturally express or compute over probabilities or stochastic pro-

cesses. Such languages include Church [26] and ProbLog [22]. Other efforts include existing languages such as BUGS [32] and Stan [41], and libraries such as Factorie [35] and Infer.NET [36].

A number of dataflow platforms exist that could be used for data analytics: Hadoop [44], Spark [46], DryadLinq [45], or Flink [6]. Of such platforms, Microsoft's Azure Data Lake Analytics is closely related to our efforts. Azure Data Lake offers a declarative programming language called U-SQL with a very tight C# binding. GraphLab [31] is also related in that it aims to support distributed machine learning.

BUDS bears some resemblance to mathematical programming languages such as R and MATLAB. Notable efforts aimed at scaling such languages include Ricardo [21] and Riot [47].

There have been some notable recent efforts at designing declarative systems for ML [12]. MADLib [27] is a set of ML algorithms implemented using SQL. Feng et al. [24] considered how a database system can be used to perform efficient incremental gradient descent. They describe how a large class of learning algorithms can be written as a specific type of user-defined aggregate. GLADE is another effort along these lines [24]. GALDE specifically targets a distributed environment, like BUDS, and has support for multidimensional arrays, like BUDS. Because both of these systems offer SQL interfaces, the resulting computations can be optimized and executed by the database system.

We have briefly discussed enhancements to SimSQL to provide vector and matrix support (see [33] for more detail). Others have also considered to incorporate array data types into relational systems. For example, PostgreSQL has support for multidimensional, variable-length array data types [4] that can easily be used to store vectors and matrices, and Oracle has the `UTL_NLA` package for linear algebra operations over its `VARRAY`s [3]. The key difference however, is that SimSQL has support for vectors and matrices, *not* arrays. Unlike simple arrays, the SimSQL optimizer is aware of the semantics of relational algebra. It knows, for example, that a multiplication of an $n \times m$ matrix and an $m \times p$ matrix is an $n \times p$ matrix, and can use this fact during optimization.

Some of the most closely related work are the efforts in array databases. For a survey, see Rusu and Cheng [40]. Qin and Rusu considered the problem of computing a dot product between a very large, sparse matrix and a set of dense vectors in a database-like environment [39]. Key database systems include SciDB [13] and SciHadoop [14]. The lack of structures such as vectors and matrices has been identified as a key reason behind the limited acceptance of relational databases in scientific applications [34]. Previous work from the database literature aimed at integrating such structures into the query language and query processing began with the development of the Nested Relational Calculus for Arrays [30] which allowed for a high-level query language based on the syntax of comprehensions [15]. In fact, comprehensions were highly influential in the design of the BUDS language. Comprehensions are closely related to relational algebra and SQL [42], and can be translated into SQL easily, as shown by approaches like the RAM algebra [43, 20].

8. CONCLUSIONS

We have presented BUDS as an example of a domain-specific language for Bayesian ML, and described techniques for compiling and executing BUDS codes on a relational database system. We considered some of the changes that need to be made to a relational database system to make it an appropriate platform for large-scale ML. Experimental results show that the BUDS optimized implementations have competitive performance compared to the hand-coded and hand-optimized SQL implementations.

Future Work. There are a few obvious avenues for future work.

Heuristic Optimization. Cost-based optimization in BUDS requires repeated calls to the SimSQL optimizer. Each one of those calls to the optimizer is expensive—taking 3 to 30 seconds. Further, this requires that BUDS be compiled to a platform that *has* an optimizer. Dataflow platforms typically do not have such optimizer. Thus, it would be valuable to design a heuristic or rule-based optimizer.

Portability to Other Platforms. Could BUDS easily be ported to other platforms? Other systems with cost-based optimizers (relational databases, or dataflow platforms such as Microsoft's Azure Data Lake) have support for arrays or user-defined types. These could be used to provide physical representations for BUDS data structures. The biggest hurdle of porting to other platforms would be imitating the "join and co-group" pattern embodied by SimSQL's VG function interface. In this pattern, for each record in a so-called *model table*, a set of parameters are prepared via a series of subqueries (the "join"s). Then the parameter sets associated with each record in the model table are fed into a table-function-based UDF (the "co-group"). This pattern is difficult to implement efficiently, in a generic fashion, without actually adding specially-coded implementation strategies directly into the underlying platform. The problem is that the total amount of data sent into the UDF is often huge. Handling this efficiently requires careful physical optimization to ensure that after the subqueries are executed, the results are sorted/hashed in a way that the co-group can be implemented by simply pipelining the parameters into the UDF.

9. ACKNOWLEDGEMENTS

Material in this paper has been supported by the NSF under grant nos. 1355998 and 1409543, and by the DARPA MUSE program.

10. REFERENCES

[1] Azure data lake. https://azure.microsoft.com/en-us/services/data-lake-analytics/. Accessed Oct 22, 2016.

[2] Mahout samsara. https://mahout.apache.org/users/environment/out-of-core-reference.html. Accessed Oct 22, 2016.

[3] Oracle utl_nla. https://docs.oracle.com/cd/B19306_01/appdev.102/b14258/u_nla.htm#CIABEFIJ. Accessed Oct 22, 2016.

[4] Postgresql arrays. http://www.postgresql.org/docs/9.1/static/arrays.html. Accessed Oct 22, 2016.

[5] M. Abadi et al. Tensorflow: Large-scale machine learning on heterogeneous systems, 2015. *Software available from tensorflow.org*.

[6] A. Alexandrov et al. The Stratosphere platform for big data analytics. *VLDBJ*, 23(6):939–964, 2014.

[7] A. Alexandrov et al. Implicit parallelism through deep language embedding. In *SIGMOD*, pages 47–61, 2015.

[8] C. Andrieu, N. De Freitas, A. Doucet, and M. I. Jordan. An introduction to MCMC for machine learning. *Machine learning*, 50(1-2):5–43, 2003.

[9] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs. In *PODS*, pages 1–15, 1985.

[10] J. Bergstra et. al. Theano: Deep learning on gpus with python. In *BigLearning Workshop*, 2011.

[11] M. Boehm, A. V. Evfimievski, N. Pansare, and B. Reinwald.

```
model HMM {
  data {
    w: range(words);
    d: range(docs);
    t: range(topics);
    priorT: array[t] of real;
    priorW: array[w] of real;
    X: array[d] of array[w] of integer;
  }

  var {
    trans: array[t,t] of real;
    emits: array[t,w] of real;
    Z: array[d] of array[w] of integer;
    A: array[d] of array[t,t] of integer;
    B: array[d] of array[t,w] of integer;
    Asum: array[t,t] of integer;
    Bsum: array[t,w] of integer;
  }

  init {
    for (i in 1:d) {
      Z[i] <- categoricalHMMStart(X[i], priorT);
    }
  }

  for (i in 1:d) {
    Z[i] <- categoricalHMM(X[i], Z[i], trans,
      emits);
    A[i] <- getTransCounts(Z[i], priorT);
    B[i] <- getEmitCounts(Z[i], X[i], priorT,
      priorW);
  }

  Asum <- sum(A);
  Bsum <- sum(B);

  for (i in 1:t) {
    trans[i] <- dirichletWithPrior(priorT,
      Asum[i]);
    emits[i] <- dirichletWithPrior(priorW,
      Bsum[i]);
  }
}
```

Figure 5: BUDS specification for text hidden Markov model learning.

Declarative machine learning-a classification of basic properties and types. *arXiv preprint arXiv:1605.05826*, 2016.

[12] V. R. Borkar et al. Declarative systems for large-scale machine learning. *IEEE DEB*, 35(2):24–32, 2012.

[13] P. G. Brown. Overview of SciDB: large scale array storage, processing and analysis. In *SIGMOD*, pages 963–968, 2010.

[14] J. B. Buck, N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. Brandt. SciHadoop: Array-based query processing in Hadoop. In *ACM SC*, page 66, 2011.

[15] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23:87–96, 1994.

[16] Z. Cai, Z. J. Gao, S. Luo, L. L. Perez, Z. Vagena, and C. Jermaine. A comparison of platforms for implementing and running very large scale machine learning algorithms. In *SIGMOD*, pages 1371–1382, 2014.

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

[18] S. Chaudhuri and V. Narasayya. Microsoft index turning wizard for SQL Server 7.0. In *SIGMOD*, volume 27, pages 553–554, 1998.

[19] E. F. Codd. A data base sublanguage founded on the relational calculus. In *Proceedings of the 1971 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control*, pages 35–68. ACM, 1971.

[20] R. Cornacchia, S. Héman, M. Zukowski, A. P. Vries, and P. Boncz. Flexible and efficient IR using array databases. *VLDBJ*, 17(1):151–168, Jan. 2008.

[21] S. Das, Y. Sismanis, K. S. Beyer, R. Gemulla, P. J. Haas, and J. McPherson. Ricardo: integrating R and Hadoop. In *SIGMOD*, pages 987–998, 2010.

[22] L. De Raedt, A. Kimmig, and H. Toivonen. ProbLog: A probabilistic Prolog and its application in link discovery. In *IJCAI*, volume 7, pages 2462–2467, 2007.

[23] J. Dean et al. Large scale distributed deep networks. In *NIPS*, pages 1223–1231, 2012.

[24] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a unified architecture for in-rdbms analytics. In *ACM SIGMOD Conference*, pages 325–336. ACM, 2012.

[25] 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*, pages 231–242, 2011.

[26] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: a language for generative models. In *Proc. UAI*, 2008.

[27] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, et al. The MADlib analytics library: or MAD skills, the SQL. *VLDB*, 5(12):1700–1711, 2012.

[28] F. Hueske, M. Peters, M. J. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas. Opening the black boxes in data flow optimization. *Proceedings of the VLDB Endowment*, 5(11):1256–1267, 2012.

[29] B. Kågström and C. F. Van Loan. *GEMM-based level-3 BLAS*. Cornell Theory Center, Cornell University, 1991.

[30] L. Libkin, R. Machlin, and L. Wong. A query language for multidimensional arrays: Design, implementation, and optimization techniques. In *SIGMOD*, pages 228–239, 1996.

[31] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein. GraphLab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

[32] D. Lunn, D. Spiegelhalter, A. Thomas, and N. Best. The bugs project: Evolution, critique and future directions. *Statistics in medicine*, 28(25):3049–3067, 2009.

[33] S. Luo, Z. J. Gao, M. Gubanov, L. L. Perez, and C. Jermaine. Scalable linear algebra on a relational database system. In *ICDE*, 2017 (to appear).

[34] D. Maier and B. Vance. A call to order. In *PODS*, pages 1–16, 1993.

[35] A. McCallum, K. Schultz, and S. Singh. Factorie: Probabilistic programming via imperatively defined factor graphs. In *NIPS*, pages 1249–1257, 2009.

[36] T. Minka, J. Winn, J. Guiver, and D. Knowles. Infer.NET 2.5, 2012. Microsoft Research Cambridge. http://research.microsoft.com/infernet.

[37] T. Park and G. Casella. The Bayesian Lasso. *JASA*, 103(482):681–686, 2008.

[38] I. Pohl. *First results on the effect of error in heuristic search*.

Edinburgh University, Department of Machine Intelligence and Perception, 1969.

[39] C. Qin and F. Rusu. Dot-product join: An array-relation join operator for big model analytics. *arXiv preprint arXiv:1602.08845*, 2016.

[40] F. Rusu and Y. Cheng. A survey on array storage, query languages, and systems. *arXiv preprint arXiv:1302.0103*, 2013.

[41] Stan Development Team. Stan: A C++ library for probability and sampling, version 2.1, 2013.

[42] V. Tannen, P. Buneman, and S. Naqvi. Structural recursion as a query language. In *DBPL*, pages 9–19. Morgan Kaufmann Publishers Inc., 1992.

[43] A. R. van Ballegooij. RAM: a multidimensional array DBMS. In *Conf. on Current Trends in DB Tech.*, pages 154–165. Springer-Verlag, 2004.

[44] T. White. *Hadoop: the definitive guide*. O'Reilly Media, Inc., 2009.

[45] Y. Yu et al. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, volume 8, pages 1–14, 2008.

[46] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *USENIX HotCloud*, pages 1–10, 2010.

[47] Y. Zhang, H. Herodotou, and J. Yang. RIOT: I/O-efficient numerical computing without SQL. In *CIDR*, 2009.

APPENDIX

A. SAMSARA AND SYSTEMML

In this section, we detail our experiences implementing the Bayesian Lasso and HMM learning on top of Mahout Samsara [2] and SystemML [25]. Both of these languages/platforms are often referred to as being declarative, and so our experiences using both platforms may be helpful in illustrating how BUDS compares to the state-of-the-art for declarative machine learning.

A.1 Mahout Samsara

Mahout provides a vector- and matrix-based algebraic environment called *Samsara* that supports R-like computations. Mahout Samsara has a close binding with Spark. The key abstraction in Mahout Samsara is the *Distributed Row Matrix* (DRM), which is a matrix partitioned by rows and stored across multiple machines. From a programmer's point of view, the multiplication of two DRMs $\mathbf{A}$ and $\mathbf{B}$ is as simple as $\mathbf{A}$`%*%`$\mathbf{B}$.

Mahout Samsara code for the Bayesian Lasso is given in Figure 6. As most computations in the Bayesian Lasso can be expressed in a linear algebraic form (such as matrix multiplication, matrix-vector multiplication and vector multiplication), the Mahout Samsara code is very similar to the BUDS specification. The non-declarative dataflow code at the beginning is used only to load and prepare the data. Mahout Samsara, like SystemML, performs algebraic logical optimization as well as physical optimization for linear algebra. We note, however, that the SystemML developers assert that Mahout Samsara is not particularly declarative since decisions on data layout and distributed/local matrices are made by the programmer [11], and hence the system does not feature data independence (the ability of different execution plans to be chosen depending upon the data characteristics).

When we executed the Bayesian Lasso code on the five machine cluster we used to execute BUDS, we found that each iteration took 202 seconds, with an initialization of 570 seconds.

```
val response = sdc.textFile("response.tbl")
    .map{line => (line.split('|')(0).toInt,
      line.split('|')(1).toDouble)}
    .map(row => row._1 -> dvec(row._2))
    .asInstanceOf[DrmRdd[Int]]
val y = drmWrap[Int](response)

val regressor = sdc.textFile("regressor.tbl")
    .map{line => (line.split('|')(0).toInt,
      line.split('|')(1).split(',')
      .map(_.toDouble))}
    .map(row => row._1 -> dvec(row._2))
    .asInstanceOf[DrmRdd[Int]]
val X = drmWrap[Int](regressor)

val yy = y - y.colMeans().get(0)
val Gram = (X.t %*% X).collect
val Sum = (X.t %*% yy).collect.viewColumn(0)

val mu = dvec(Array.fill[Double](X.ncol)(1))
var sig = invGamma(1, 1)
var tau = invGauss(mu, 1)

val niter = 5
for(i <- 0 until niter) {
  var A = solve(Gram + diagv(tau))
  var b = multiNormal(A %*% Sum, A * sig)

  var Z = yy - X %*% b
  sig = invGamma((y.nrow - 1) / 2 + X.ncol / 2,
        (Z.t %*% Z).collect.get(0, 0) / 2
          + (b * b).dot(tau) / 2)

  val lam = 1.060047
  tau = invGauss(((lam * lam * sig)
                / (b * b)).sqrt, lam * lam)
}
```

Figure 6: Mahout Samsara code for the BL.

In our opinion, Mahout Samsara achieves the declarative ideal for Bayesian Lasso learning. However, Markov chain simulation for HMM learning on Mahout Samsara is more challenging. For HMM learning, each document is most naturally represented as a sequence of numbers, where the numbers code, in order, the sequence of words in the document. Each sequence can be a different length, depending on the length of the document. Thus, the most natural representation for a document corpus in HMM learning is as a set of sequences. Fortunately, Mahout Samsara's DRM is essentially a wrapper for a Spark RDD of row vectors with type `[RowID, Vector]`, and so it can be used to store the corpus.

Specifying the required computation is difficult, however. Mahout Samsara is essentially a distributed matrix system, and learning an HMM does not map naturally to matrix computations. In particular, it is necessary to loop through all of the words in a document, in sequence, and re-assign each word to a particular topic in the current model. This requires various statistical computations, culminating with a sample from a multinomial distribution to perform the topic assignment, whose results must be aggregated via a series of somewhat complicated computations.

After some time spent investigating various ways to achieve this using Mahout Samsara, we identified two different options. One is to use Mahout Samsara's `iterator()` method over DRMs, which returns an iterator capable of looping over all of the rows in the matrix. However, as far as we can tell, this iterator is not meant to permit parallel execution. It appears to collect the entire DRM into the RAM of the driver machine where iteration takes place. This is problematic for at least a couple of reasons. First, the corpus

```
invGamma = externalFunction(...)...
invGaussian = externalFunction(...)...
multiNormal = externalFunction(...)...

X = read("test_data/xb.bin", format="binary")
y = read("test_data/yb.bin", format="binary")
y_avg = avg(y)
y = y - y_avg

# compute the matrix X'X, and X'Y
XX = t(X) %*% X
XY = t(X) %*% y

# number of data points and number of features
n = nrow(X)
m = ncol(X)

shape_prior = 1.0
scale_prior = 1.0
mean_prior = matrix(1.0, rows=1, cols=m)
sigma2= invGamma(shape_prior, scale_prior)
tau= invGaussian(mean_prior, shape_prior)

niter = 5

for (i in 1:niter) {

  A = XX + diag(t(tau))
  A_inv = inv(A)
  mu = A_inv %*% XY
  covariance = A_inv * sigma2
  beta = multiNormal(t(mu), covariance)
  remain_sum1 = (t(y) - beta %*% t(X))
    (y - X %*% t(beta)) / 2.0
  remain_sum2 = (beta * beta) %*% t(tau) / 2.0
  scale_m = 1.0 + remain_sum1 + remain_sum2
  scale = as.scalar(scale_m[1,1])
  shape = 1.0 + (n-1.0)/2.0 + m/2.0
  sigma2 = invGamma(shape, scale)
  tau_mu = sqrt(sigma2 / (beta * beta))
  tau = invGaussian(tau_mu, 1.0)
}
```

Figure 7: SystemML code for BL.

```
# generate new states for the words
parfor (id in 1:doc_num) {

  labels = matrix(0, rows=1,
    cols=as.scalar(doc[id, 1]))
  workProbs = matrix(0.0, rows=1, cols=topic_num)

  # for the first word
  parfor (i in 1:topic_num) {
    workProbs[1, i] = start_prob[1, i] *
        trans[i, as.scalar(topics[id, 1])] *
        emis[i, as.scalar(doc[id, 2])]
  }
  tmp_first = categorical(workProbs[1, ])
  labels[1, 1] = tmp_first

  # for the middle words
  for (j in 2:as.scalar(doc[i, 1]) - 1) {
    parfor (i in 1:topic_num) {
      workProbs[1, i] = trans[
        as.scalar(labels[1, j - 1]), i] *
        trans[i, as.scalar(topics[id, j + 1])] *
        emis[i, as.scalar(doc[id, j + 1])]
    }
    tmp_middle = categorical(workProbs[1, ])
    labels[1, j] = tmp_middle
  }

  # for the last word
  parfor (i in 1:topic_num) {
    workProbs[1, i] =  trans[as.scalar(
      labels[1, as.scalar(doc[id, 1]) - 1]), i] *
      emis[i, as.scalar(
      doc[id, as.scalar(doc[id, 1]) + 1])]
  }
  tmp_last = categorical(workProbs[1, ])
  labels[1, as.scalar(doc[id, 1])] = tmp_last

  parfor (i in 1:ncol(labels)) {
    topics[id, i] = labels[1, i]
  }
}
```

Figure 8: Snippet of the SystemML code for HMM inference. This code is written in SystemML's *DML* language, and updates the state associated with each word in each document in the data set.

can easily be too large to reside in the RAM of the driver machine. And second, there is no distributed execution of the learning.

The second option would be to use the DRM's `mapBlock()` method, which performs a map-style iteration over the matrix blocks underlying a DRM. This can be used to produce a second DRM that lists the assignment of words to topics for each document. However, it is next necessary to perform a set of rather complicated aggregations over the original data matrix and the resulting topic-assignment matrix. For example, for each adjacent pair of topics, the number of times that `topicA` transitions to `topicB` must be computed. In the BUDS code of Figure 5, this is accomplished via the `getTransCounts` user-defined function that maps the topic assignments for each document to a matrix of transitions, followed by a `sum` over the resulting set of transition count matrices. However, in Samsara, since a `mapBlock` operation cannot create a matrix of transition matrices from the matrix of topic assignments, this is difficult to realize.

The only real option left is to use the fact that Samsara is implemented on top of Spark, and that it allows one to write Spark RDD operations over DRMs. Careful tuning and implementation in this way results in a highly-performant code. On our five-machine cluster and data set, initialization takes 222 seconds, and later iterations take 1,236 seconds. This is again somewhat faster than BUDS. On the negative side, few programmers would describe the resulting implementation as anything but a carefully hand-tuned Spark code that is not particularly declarative.

A.2 SystemML

SystemML provides a high-level language syntax for scalable machine learning. It offers two languages: an R-like language called *DML*, and a Python-like language called *PyDML*. It also supports multiple execution modes such as *Spark MLContext*, *Spark Batch*, *Hadoop Batch*, and so on. When performing large-scale linear algebra computations, SystemML uses cost-based optimization to generate very efficient low-level execution plans.

We begin with our implementation of the Bayesian Lasso on top of SystemML, depicted in Figure 7. The responses and regressors are stored as SystemML matrices. The major computations are conducted through a series of linear algebra operations. External Java functions provide random sampling functionality. Since this is mostly a linear algebra computation, the SystemML code appears very close to both the BUDS code and the Samsara code. We have tested this code on the same dataset and cluster as we did for the

```
parfor (i in 1:doc_num) {
  parfor (j in 1:as.scalar(doc[i, 1])) {
    tmp = categorical(topic_prior[1, ])
    topics[i, j] = tmp
  }
}
```

Figure 9: Snippet of the SystemML code for HMM initialization. This code is written in SystemML's *DML* language, and samples the initial state associated with each word in each document in the data set.

BUDS implementation. The initialization time is 57 seconds, and the running time is 90 seconds per iteration. This result indicates that SystemML is very competitive for linear algebra computations.

However, just as in the case of Mahout Samsara, it becomes difficult to write declarative code on top of SystemML as soon as it is used to perform a computation that does not map nicely into vectors and matrices. HMM learning was particularly troublesome because if one views a document corpus as a matrix—the fundamental data type supported by SystemML—each row in the matrix will have a different length, corresponding to the length of the document. Since this is not supported directly by SystemML, we have to use a bit of a trick. We store all documents in a matrix, where the number of columns in the matrix is `max_length` + 1 (`max_length` is the maximum length over all documents). The extra column is used to store the length of each document so that we only visit the valid entries.

We give just a subset of the SystemML code for the Bayesian HMM in Figure 8 because the entire code is quite long. This subset gives the loop that walks through the various documents, updating the hidden topic used to produce each word in the document. Since SystemML supports comprehensions via the `parfor` statement, we use `parfor` to execute the statements in parallel.

Unfortunately, we had a difficult time using the `parfor` construct. When we tuned the parameters for the `parfor` loops (e.g., multi-core execution, map-reduce execution, data partition, task partition, etc.), we found that the system would either generate too many map-reduce tasks, or produce a map-reduce task which was too large. Even for the initialization of the hidden topics (see Figure 9), we were unable to produce an execution that could finish in five hours (after which time, we killed the computation). We also had difficulties similar to what we experienced using Mahout Samsara computing the topic-to-topic transition statistics, as well as statistics describing how often a topic emits a word. We found that these would have to be calculated by a linear scan of all documents, because SystemML does not provide the aggregation operation for the matrices produced in the parallel statements.

Just like Samsara, SystemML does have a Spark binding, and so we could have written a code very similar to the Mahout Samsara code–essentially a Spark code storing data in SystemML matrices. But this would render SystemML essentially just another way to access Spark's RDD operations.

A.3 Summary

Both Mahout Samsara and SystemML codes look a lot like BUDS codes for the Bayesian Lasso, and they actually give better performance on this task. This is not surprising, as both of these systems are effectively highly performant, scalable linear algebra systems, as the Bayesian Lasso learning is a linear algebra task. However, HMM learning was very different. Since HMM learning does not map nicely to simple linear algebra computations, and requires transformations such as mapping a set of variable-length vectors (the document corpus) to a set of matrices (the topic-to-topic transition counts), it is difficult to avoid simply writing RDD transformations using each system's Spark binding. This illustrates what we believe is the most significant distinction between BUDS and these two systems: while BUDS supports declarative computations over complex data structures (mixtures of maps, sets, and arrays), Mahout Samsara and SystemML seem, at present, to be better suited to linear algebra tasks.