The BUDS Language for Distributed Machine Learning

Anonymous

ABSTRACT

In this paper, we describe BUDS, which is a specialized, declarative domain specific language (DSL) for suppoting implementation of large-scale, distributed, Bayesian machine learning algorithms on top of a relational database system. There are two key optimization challenges that must be addressed when compiling BUDS so that it can run on top of a database. First, the system must determine how to represent domain-specific data structures (vectors, matrices and maps in the case of BUDS) in the relational system. Second, the system must perform domain-specific re-writes. We show that in the case of BUDS, it is possible to declaratively specify quite complex machine learning computations in just a few lines of simple code, and that these computations can be executed efficiently in a distributed fashion over large data sets.

1. INTRODUCTION

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

1. **The whiteboard step**–First, do the math necessary to define the model and derive the learning algorithm.
2. **The small data prototype**–Next, build prototype of the model and 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**–Finally, build a robust, distributed or parallel implementation of the learning algorithm and apply it to a large, production data set.

In practice, this workflow can require a tremendous amount of effort, some of which may be unavoidable. Developing a model and a learning algorithm is something of a black art, requiring experience, intuition and trial-and-error. Moving from the whiteboard to a small data prototype can require an iterative process, where the model and learning algorithm are refined over time.

However, not *all* of the effort often required to realize this workflow appears to be absolutely necessary. Moving to a distributed or parallel implementation of a large-scale learning algorithm from a Matlab-based prototype often requires a frustrating amount of effort, even using a dataflow platform such as Hadoop or Spark. Much of this effort is related to the the programmer working as something of a "human optimizer." A programmer must make 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 operation been run first, the join would have essentially been costless.

These may be examples of obviously poor programming choices, but the fact is that getting a statistical computation to run on a very large data set is difficult. 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 (such as the interface exported by Hadoop and Spark) on the characteristics of the data it operations 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 [] and eventually the declarative subset of SQL—two inventions that began thirty years of dominance for the relational model.

DSL or General-Purpose Language? One might reasonably ask: Is the solution to overcoming these problems adding a database-style, cost-based optimizer to existing dataflow platforms?

The answer appears to be no. Dataflow platforms such as Spark, Hadoop, Flink, DryadLinq, and others are essentially APIs embedded in a host programming language such as Python or Java. On one hand, dispensing with a domain-specific language (or "DSL")—historically SQL—is a key selling point of such platforms. One of the major drawbacks of SQL-based database systems for any sort of domain-specific computation is that integration with full-featured programming languages, and hence access to standard mathematical, statistical, and scientific libraries, has always been difficult. On the other hand, dispensing with a DSL means that much of the logic associated with the computation is embedded in the host language and opaque to any optimizer. For example, it is not possible for an optimizer to understand that a set of matrices in the host language could be normalized to a set of row vectors managed by the platform.

The BUDS Programming Language. Motivating the work described in this paper is the belief that ultimately, to cover the wide variety of applications in the data analytics space, a set of highly-specialized declarative DSLs for data analytics are needed. In this paper, we describe the BUDS programming language. BUDS is a specialized, statistically-oriented DSL for declaratively specifying a Markov chain whose state consists of millions or even trillions of values, and may require terabytes to store. It is targeted specifically at making the implementation of a statistical code for a large data set a painless process.

At first glance, BUDS looks something like a variant of R or MATLAB, with the difference being that a BUDS program is essentially a list of dependencies among variables; these dependencies declaratively specify a Markov chain (in this sense, BUDS shares similarities with other DSLs for stochastic simulation such as BUGS [**?**] and Stan [**?**]). The BUDS compiler then figures out the best way to implement that Markov chain using an underlying relational database system. The key benefit of BUDS is that it is exceedingly simple to produce machine learning programs that can run on hundreds of machines using just a few lines of BUDS code. BUDS Markov chain specifications are designed to look almost identical to the mathematics that one might find written in an academic paper, with data represented as vectors and matrices. All of the details regarding how the computation actually works are absent from the BUDS specification—they are figured out by the system. Another key benefit of BUDS is that it is quite easy to implement user-defined functions for BUDS that allow core computations to be written in C/C++. Crucially, BUDS itself automatically solves many of the type mismatch and marshaling/unmarshaling complications that plague user-defined functions writing in general-purpose programming languages written for classical database systems.

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 $\Pr[X_{i+1} = x_{i+1}|x_0, x_1, ..., x_i] = \Pr[X_{i+1} = x_{i+1}|x_i]$.

The reason that we are interested in declaratively specifying and simulating a Markov chain with a very large state is that Markov chin simulation is the most fundamental method for learning a statistical model in Bayesian machine learning (ML), where it is known as Markov chain Monte Carlo, or MCMC [**?**]. Bayesian learning requires the characterization of a posterior distribution. If the posterior distribution is complex and high-dimensional, this can be impossible to do analytically, and one common method for understanding the posterior is to draw random samples from it. MCMC methods for Bayesian ML work by defining a Markov chain such that it is possible to draw a sample from the posterior by simulating the defined chain for a number of transitions, and then taking the state at the end of the simulation as a sample from the posterior—that is, they define a Markov chain whose *stationary distribution* is precisely the target posterior.

We note that while our focus is on the description and simulation of Markov chains, BUDS is also very useful for specifying and running non-random machine learning or statistical algorithms—after all, deterministic, iterative optimization algorithms such as EM [**?**] can be viewed as Markov chain simulations where there is only one possible value for x_{i+1} given x_i.

Our Contributions. Our contributions in this paper are as follows:

1. We describe the BUDS language in detail. While BUDS is specifically geared towards Bayesian ML, it is a good example of the sort of highly specialized declarative DSL for data analytics that should exist for other specialized domains.

2. We consider the problem of how to compile programs written in BUDS into computations that can be run on top of SimSQL [**?**], which is a distributed database system. We focus on two key technical questions that arise when compiling BUDS for SimSQL. First: How are data structures in BUDS represented efficiently as tuples, vectors, and matrices in SimSQL? Second, how are operations BUDS properly translated in the target DSL so as to provide for an efficient execution?

3. We have fully prototyped the BUDS language as well as our proposed translation framework. We show that BUDS can provide good performance for a variety of machine learning computations on a distributed compute cluster.

Paper Roadmap. In the next section of the paper, we give a few examples of BUDS programs, discuss the general philosophy of the BUDS translation process, and also describe SimSQL's SQL, which is the target of the BUDS translation process. Section 3 of the paper describes the BUDS language in depth. Section 4 describes the compilation process in detail. Section 5 benchmarks a few BUDS programs, and Section 6 presents a review of the related work. Section 7 concludes the paper.

2. BUDS OVERVIEW

In this section, we give a simple example of a BUDS specification.

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 is 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

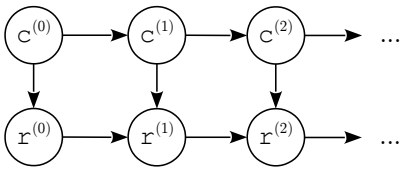

Figure 1: Graph of variable dependencies for the cities-restaurants model, iterations $i = 0, 1, 2$.

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;
}
```

In the above code, the `range` type is used to associate a data domain with an integer constant. For example, the domain of `cities` is identified with integer keys from the set $\{1, 2, \ldots, \mathtt{n}\}$. Users can also describe structures over such domains, such as the matrix `D`, defined as an `array` (here, a dense two-dimensional structure) over the domains `cities` and `restaurants`.

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. As is standard in probabilistic models, 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 [**?**], an idea that we will make heavy use of in BUDS.

2.2 Another BUDS Example

We now give a complete BUDS implementation for an actual Bayesian ML algorithm. The Bayesian Lasso [**?**] 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 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

This section describes the BUDS language in more detail.

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`

```
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.

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[1] 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. We might represent such a data set as:

```
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

Variable Expressions. The expression on the left-hand side of an `<-` assignment 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 instance, separately applying an assignment expression on the rows of a matrix or the entries of a vector, as in the case of the Lasso vector $\boldsymbol{\tau}$ shown above. The model statement for sampling $\boldsymbol{\tau}$ during the remaining generations would be as follows:

```
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

[1]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.

denote a "loop" in the sequential, procedural-programming sense; rather, it denotes a series of 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 cells of a compound variable inside the "loop" are invalid. Let us return to the cities-restaurants example from Section 2.1, where a user might be tempted to denote the conditioning of 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);
}
```

The above code does not compile in BUDS, since it is not possible to assign a variable to an expression more than once in a loop. Furthermore, given that the order of assignment statements is inconsequential in BUDS, it is not possible to ensure that the assignments on `tempD` happen "before" `r[j]` is generated.

There is one case where the use of the temporary variable `tempD` is possible in such a loop: as long as each execution of the loop updates a distinct portion of a compound type. For example:

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

This is valid since `tempD` and the loop are both defined on the range `k` (or, alternatively, in the domain of `individuals`).

Recursion. Given the declarative semantics of BUDS, variables are immutable and used to represent the result of a specific computation, rather than the state of the program. As such, any given variable can only be assigned to a single expression by way of a model statement. Moreover, model statements can be written in any order and that, with the exception of initialization statements, circular references between variables are permitted—in fact, such references are essential, as they describe the recursive structure of the computation. 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.

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 $*$ 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 $d_1, d_2, \ldots, d_{n-1}$. 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[m]` structure where each entry `X[i]` equals $\sum_j$ `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 [**?**]. 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 in a canonical form that is easy to manipulate during the latter phases of the translation process. Comprehensions are also central to the BUDS model of parallelism. Essentially, the range or 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. For example, the computation of the vector `t` from the Bayesian Lasso is expressed so that the computation of each entry `t[j]` can be done independently by invoking the inverse-Gaussian function.

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 (this is one of the models we implement using BUDS in the experimental evaluation associated with the paper). 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. THE BUDS TRANSLATOR

A fundamental question that one must ask when deciding how to execute a language such as BUDS is: What is the abstract machine that will execute a BUDS program?

4.1 Design Considerations

At the highest level, it seems that some sort of relational algebra engine is the correct execution enivonment for BUDS. A BUDS program essentially consists of a set of joins, selections, projections, and aggregations along with external functions that must be run until convergence to a fixpoint.

This leaves the question of *how* to obtain and execute the correct relational algebra program. Modern dataflow systems such as Hadoop, Spark, Flink, and so on basically expose relational algebra interfaces, and hence serve as a possible execution environment. Building a compiler and execution environment directly on top of one of these systems would require:

1. Translation into relational algebra;
2. optimization of the algebra;
3. execution of the algebra.

Algebraic optimization of a relational agebra computation is exactly what a database query optimizer does. Engineering a modern database query optimizer is difficult and time-consuming task, and so it seems to make sense to implement BUDS on top of (at least the backend of) a relational database system rather than a general-purpose dataflow engine.

Once this decision is made, there is yet another question to ask: shoud BUDS be translated 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, rather than relational algebra. The reason is that 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 rewriting []. Presumably, the translation process itself will be a bit slower using SQL as an intermediate (after all, lexing and parsing SQL is not free) but this cost can be mitigated by directly generating an SQL parse tree rather than text, and optimization and execution are likely to be so expensive anyway that the additional cost will be negligable.

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

- 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.
- 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.

Performing this translation is a surprisingly difficult task. There are two key questions that any such translation must address:

1. First, there is typically going to be a mismatch between the data representation in the source and target SQL database. For example, in BUDS, data are stored in arrays, maps and lists, whereas in SimSQL, data are stored in relations. Thus, the compilation and optimization process must choose a suitable and efficient data representation.
2. Second, because SQL is far more general than BUDS, there are likely going to be operations and data structures in BUDS whose semantics will be opaque to the target compiler and optimizer, and thus there are going to be important re-writes and implementation choices that must be taken care of by the translation process.

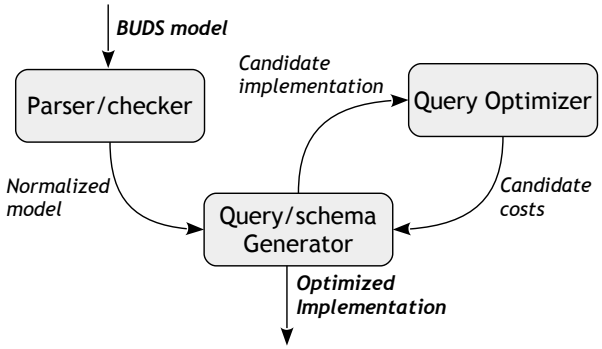

Figure 3: Architecture of the BUDS translator.

Our central idea is to explore the space of alternative implementations in the target DSL using an A*-style search algorithm [**?**]. That is, we first translate the source DSL specification into a target DSL code whose semantics are equivalent, and then we employ a series of transformations that move from one target implementation to another. When an alternative implementation is generated, it is sent to the target DSL's optimizer for cost analysis. This idea is reminiscent of the tactic pioneered by the developers of Microsoft SQLServer's AutoTune wizard [**?**]. The resulting cost is used to guide the search over possible target language implementations.

The different phases of the BUDS translation process are depicted in Figure 3. During the first phase, the code is parsed and checked for syntactic correctness, and the expressions in each model statements are checked for type consistency. Thereafter, variable references in the model are checked for circularity and the initialization statements are verified to confirm that they suffice for starting off the chain. Once the correctness of the model has been verified, the translator "normalizes" the model by removing temporary variables and blocks, leaving only stochastic assignments. The normalized model consists of:

- The set of variables corresponding to the base dataset, together with their data types as described in the *data* section
- The set of variables associated with stochastic assignments, together with the expression trees that define how to initialize and update their values
- The dependency graph outlining the order in which variables are to be updated each generation.

During the second phase, the translator traverses the space of possible base dataset schemas for the constant variables and possible SQL implementations for sampling and updating the values of the variables. 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.

5. THE TARGET PLATFORMI: SIMSQL

While in theory we could have chosen any parallel database system as the BUDS execution platform, we decided to choose the SimSQL system [**?**] as the BUDS target. There are two main reasons for this. First is SimSQL's support for a very flexible type of user-defined function called a *VG function*, which allows relatively complex contructions taking multiple tables as input, and producing multiple tables as output. Second is SimSQL's support for recurively-defined tables, which make it a natural target for fixpoint computations. Choosing another platform would have required building such capabilities (or similiar ones) on top of the system.

One thing missing from SimSQL is native support for vectors and matrices, which is crucial to producing efficient statistical codes—enociding such data structures as relations generally results in poor performance, as our experiments will show.

In the remainder of this section, we give a brief overview of SimSQL's VG function interface, as well as its support for recursive computations. We breifly describe our SimSQL extensions for vectors and matrices; a full description can be found in our technical report on the subject [].

5.1 SimSQL's SQL

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 Categirical 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 current 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`).

Vectors and Matrices. For the BUDS-to-SQL translation task, To acheive 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 ba 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, 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. 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.

Note that our extension to SimSQL does not support an array database model, nor does it support large (out-of-core) matrices and vectors. It merely supports matrix and vector data types, as well as natively-supported operations over them.

6. TRANSLATING BUDS MODELS

This section describes in detail how BUDS models are translated into SQL.

6.1 Compilation

In general, the translation process begins by parsing the input and performing semantic checks. Next, a dependency graph is created. In the case of BUDS, the translator first eliminates all `for` blocks to replace them with comprehension expressions, resulting in a model that consists entirely of simple variable-expression assignments. The translator then constructs a graph where each vertex corresponds to a model variable from the `data` and `var` sections and each edge is of the form $u \rightarrow v$ denoting a reference to variable v in the expression assigned to variable u. The result is a directed acyclic graph annotated with iteration numbers. Figure 4 shows the data dependency graph for the Bayesian Lasso. Once the dependency graph is created, depending on the translation problem,

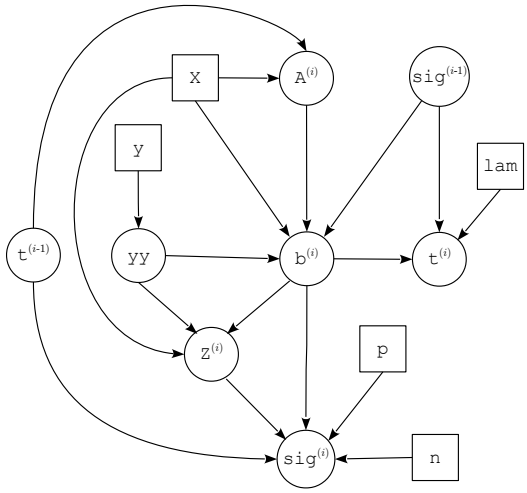

Figure 4: Data dependency graph for the Bayesian Lasso.

it may be analyzed to check for correctness. In the case of BUDS, we check if the initialization statements from the `init` section are sufficient to initialize all of the variables needed in subsequent iterations of the computation. That is, each of the variables with an $(i-1)$ super-script must be initialized. Also, there can be no cycles among variables labeled (i), since this depicts a circular dependency.

After semantic checking, we are ready to begin the translation.

6.2 Moving Among Data Representations

Programmers in BUDS choose from data structures such as arrays, matrices, maps, etc. One of th emost iportant tasks in the translation process is choosing how these data structures can be represented in the underling relational/linear model. For example, the matrix `D` from BUDS in our city-and-restaurants model can be represented in SimSQL's SQL 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 incumbant upon the tranlation framework to 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. A small subset of the BUDS-to-SimSQL `typeGraph` relation is depicted in Figure 5. This shows four possible representations for a matrix (or two-dimensional array) in BUDS.

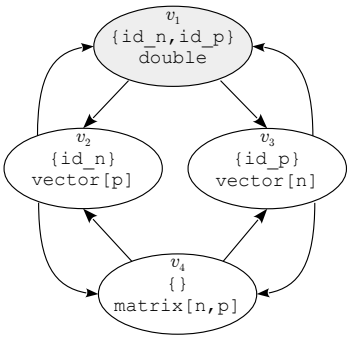

Figure 5: Type graph over possible representations of the matrix variable X.

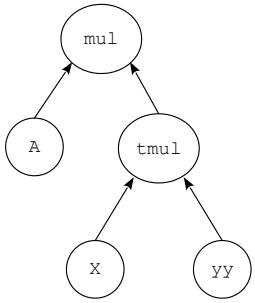

Figure 6: Expression graph for part of the Bayesian Lasso.

Each edge in this graph has an associated eSQL code template associated with it, that can be used to generate the code associated with the movement between 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 5. We have the corresponding Datalog rule (note that In Datalog rules 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):

```
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. Here is the rule for the edge from v_2 to v_1:

```
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.
```

Here, `inName + outName` is used to create a unique identifier for the table holding intermediate results. For example, the function call `xformImp(v1, v1, inName, outName)` will return the following code string:

```
CREATE VIEW inNameoutName
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;

CREATE VIEW out
AS SELECT inp.id_n AS id_n, r.id_p AS id_p,
          GETSCALAR(inp.val, r.id_p) AS val
FROM inNameoutName AS inp, responses AS r;
```

6.3 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. 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 optimial for large matrices that are small enough to fit in memory. But it will fail for very large matrices. All of these options must be considered systematicaly during search.

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 and input edges (which we will call "leaf" vertices, through we may not have a tree) represent data (in the case of BUDS, they are variables from the `data` and `var` sections of the program) and 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:

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

The corresponding expression graph is shown in Figure 6.

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 in that they provide code for moving between types. Node 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 of different input types and produce different output types, though they are all doing the same computation. 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 we need a way to store instantiated implementations associated with out particular translation problem. The relation `imp (Type, Name, Str)` will accomplish this, storing for us in the string `Str` for all available implementations of the node with name `Name`, where the result takes the type `Type`. This relation is defined as follows. 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 avialble implementation is to do nothing, if we have a leaf node:

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

And we can possibly run a 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 this 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.

6.4 Encoding Domain Specific Optimizations

One tremendous advantage of performing optimizations on a specialized, declarative DSL such as BUDS (as opposed to a more general-purpose DSL such as eSQL) is that encoding domain-specific optimizations 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 + ";"
```

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

6.5 Implementation Details

The previous subsection described how all possible implementations for a given source DSL 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 (see the experimental section for an example of this, where for a Bayesian Gaussian Mixture Model, there were 1,214 SimSQL implementations). Thus, in practice, it is going to be necessary to add some sort of search heuristic. We have implemented an A*-style search, and found that in fact, a purely greedy algorithm works very well. The greedy search works as follows. 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

```
data {
  n: range(points);
  d: range(dims);
  k: range(clusters);
  X: array[n,d] of real;
  prMix: array[k] of real;
  prMean: array[d] of real;
  prCovar, prScale: array[d,d] of real;
  prDegrees: integer;
}

var {
  mix, ccount: array[k] of real;
  means, cmean: array[k,d] of real;
  covars, csum: array[k,d,d] of real;
  Z: array[n] of integer;
}

init {
  mix <- dirichlet(prMix);
  for (j in 1:k) {
    covars[j] <- invWishart(prScale, prDegrees);
    means[j] <- normal(prMean, prCovar);
  }
}

for (i in 1:n) {
  Z[i] <- categoricalGMM(mix, X[i], means, covars);
}

for (j in 1:k) {
  ccount[j] <- count({ Z[i] | i in 1:n, Z[i] = j });
  csum[j] <- sum({ outer(X[i] - means[j])
           | i in 1:n, Z[i] = j });
  cmean[j] <- mean({ X[i] | i in 1:n, Z[i] = j });

  covars[j] <- invWishartGMM(prScale, prDegrees,
                csum[j], ccount[j]);
  means[j] <- normalGMM(prMean, prCovar,
              covars[j], ccount[j], cmean[j]);
}
mix <- dirichlet(prMix + ccount);
```

Figure 7: BUDS specification for the Gaussian Mixture Model's MCMC algorithm.

representation. Each of those is costed, and the lowest-cost alternative is chosen as the new implementation. This is repeated until the implementation cannot be improved.

7. EXPERIMENTAL EVALUATION

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 SQL program. Our current implementation has 16 different SQL 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 SQL 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.).

In this section, we describe an experimental evaluation of the performance of the BUDS language and compiler for a set of representative Bayesian machine learning problems. The key question will be how BUDS-encoded programs compare performance-wise to programs written directly in SQL.

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).

7.1 Experimental Overview

Since BUDS is translated into SimSQL's SQL, we being by hand-coding one or two implementations of each machine learning algorithm directly in SimSQL's SQL as a baseline.

These hand-coded implementations typically include one naive implementation and one carefully-tuned implementation, subject to the constraint that our hand-coded 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).[2] We then experimentally compare the computational efficiency of these hand-coded baseline implementations with the implementation produced by the BUDS translator—both before and after the BUDS translator optimizes the generated code. Since BUDS is free to choose vector- and matrix-based implementations for the various algorithms, in the ideal case, one might hope that the BUDS codes would outperform the hand-coded implementations. At the very least, one would hope that they will be no slower than the hand-written code.

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.

7.2 Models Tested

Our experiments focus on the implementation of Markov Chain simulations for learning the following for 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 five million data points distributed across the five machines, having 1,000 regressor dimensions each.

Gaussian Mixture Model. The GMM is a standard model for unsupervised data clustering. The model assumes that the data set to be processed was generated by a mixture of Gaussians (multi-dimensional normal distributions) and the task is to recover the various components from the data. We learn a Bayesian variant of this model (see [**?**] for details). The full BUDS code for learning this model is given in Figure 7. To evaluate the various im-

[2]We choose to use tuple-based implementations as a baseline because in a previous paper [**?**] we evaluated the performance of such tuple-based SimSQL codes vis-a-vis codes written for several other platforms, such as as GraphLab and Giraph. Hence, one can use such a tuple-based baseline in conjunction with those previous results to guess how a BUDS code might compare to a code written natively on one of these platforms.

Model	Implementation	Code Lines	Run Time
BL	Naive SQL	100	00:07:28 (02:38:46)
BL	Vector/Matrix 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	Vector/Matrix 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	Vector/Matrix SQL	101	00:29:28 (00:01:13)
LDA	BUDS no-opt	31	13:54:32 (17:13:55)
LDA	BUDS opt	31	00:30:21 (00:53:58)
HMM	Naive SQL	131	08:17:07 (10:51:32)
HMM	Doc-based SQL	123	03:36:47 (00:17:51)
HMM	Vector/Matrix 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 parens is the time for the initialization iteration.

plementations, we use a 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. Note that two 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 (again, see [**?**] for more details).

Latent Dirichlet Allocation. LDA is a very standard topic model for unsupervised learning over text. The model assumes that a textual document is produced by a mixture of "topics", which are essentially vectors that control the frequency or prevalence of each word in the topic. We use a non-collapsed Gibbs sampler to learn this model (again, see [**?**] 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 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 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.

7.3 Results and Discussion

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 repeatedly chooses the best Datalog rule to fire ("best" in terms of the rule that results in the expression graphs with the lowest cost according to the SimSQL optimizer) until the expression graph can no longer be improved. Interestingly, in each case, the greedy strategy resulted in the optimal expression graph being found.

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

There are a few interesting findings. First, in every case, the SQL code produced by the BUDS compiler without optimization performs just about as well as the naive SQL code written by hand. This is to be expected: "naive" here means that the most obvious encoding of the problem has been chosen by the programmer.

In every case, the optimization process is able to arrive at a data representation and implementation that is far superior, and outperforms even the optimized, hand-coded versions (which are constrained to only use the pure tuple-based encoding of the various data structures). For example, through optimization, BUDS decides to encode set of mean vectors sent to the `categoricalGMM` function as a matrix, and the set of covariance matrices sent to this function are encoded as a relation containing tuples with matrix attributes. The cost of parameterizing this function is radically reduced because of the much smaller number of parameters. It is interesting to note that in every case, the optimized BUDS implementation was superior to the others—the one exception being the block-based GMM. 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 sort of optimization is not available to the BUDS compiler.

8. RELATED WORK

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 processes. There has been a lot of recent work in this area, including general-purpose languages such as Church [**?**] and ProbLog [**?**]. Other efforts include existing languages such as BUGS [**?**] and Stan [**?**], and libraries such as Factorie [**?**] and Infer.NET [**?**]. One key difference between these efforts and BUDS, however, it that BUDS is specifically focused on very large scale probabilistic computations. We also focus on executing on top of a declarative DSL such as SQL. These other systems are typically concerned with program synthesis and execution at a very small scale.

BUDS bears some resemblance to mathematical programming languages such as R and MATLAB. Notable efforts aimed at scaling such languages include Ricardo [**?**] and Riot [**?**]. SystemML [**?**] from IBM has a custom-designed scripting language that looks a lot like these languages, and it is also scalable.

There have been some notable recent efforts at designing declarative systems for machine learning [**?**]. Again, our work differs in that we are interested in executing on top of an existing system, rather than architecting a new one.

MADLib [**?**] is a set of machine learning algorithms implemented using SQL. This is similar in spirit to the work presented here, except that the codes in MADLib are all hand-written.

Some of the most closely related work are the efforts in array databases. 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 [**?**]. 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 [**?**] which allowed for a high-level query language based on the syntax of comprehensions [**?**]. In fact, comprehensions were highly influential in the design of the BUDS language. Some of the advantages of the syntax of comprehensions are that it allows for elegantly describing collections such as arrays and sets using queries, its close relationship with relational algebra and SQL [**?**], and that that it can be translated into SQL easily, as shown by approaches like the RAM algebra [**?**, **?**]. This latter work is probably the related work closest to our own, but there are some key differences. While that work was primarily concerned with synthesizing purely tuple-based relational codes, we are interested in searching through multiple possible translations, including native matrix and vector representations. Another is the focus in BUDS on iterative, large-scale computations, which is quite different from the focus on algebraic operations over arrays. Other notable array database systems include SciDB [**?**] and SciHadoop [**?**].

Note that there have been efforts to scale up statistical computing platforms such as MATLAB and R [**?**, **?**], and IBM's SystemML [**?**] supports a MATLAB like language for cluster computing. Crucially, these systems are not delcarative.

9. CONCLUSIONS

We have argued that the need for specialized, declarative domain specific languages (DSLs) for data analytics does not require the development of new computational platforms. Instead, we argued for leveraging the general DSLs provided by such platforms using compilation and optimization techniques. We have presented BUDS as an example of a highly-specialized DSL for Bayesian machine learning, and described techniques for compiling BUDS codes into a general data processing DSL which, in our proof-of-concept implementation, is the variant of SQL used in SimSQL for describing Markov chain Monte Carlo simulations. In our empirical evaluation, we wrote BUDS code for a variety of Bayesian machine learning algorithms, and measured the performance of the resulting SQL implementations in a distributed setting.