Cumulon: Optimizing Statistical Data Analysis in the Cloud[*]

Botong Huang
Duke University
bhuang@cs.duke.edu

Shivnath Babu
Duke University
shivnath@cs.duke.edu

Jun Yang
Duke University
junyang@cs.duke.edu

ABSTRACT

We present Cumulon, a system designed to help users rapidly develop and intelligently deploy matrix-based big-data analysis programs in the cloud. Cumulon features a flexible execution model and new operators especially suited for such workloads. We show how to implement Cumulon on top of Hadoop/HDFS while avoiding limitations of MapReduce, and demonstrate Cumulon's performance advantages over existing Hadoop-based systems for statistical data analysis. To support intelligent deployment in the cloud according to time/budget constraints, Cumulon goes beyond database-style optimization to make choices automatically on not only physical operators and their parameters, but also hardware provisioning and configuration settings. We apply a suite of benchmarking, simulation, modeling, and search techniques to support effective cost-based optimization over this rich space of deployment plans.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Statistical databases*

Keywords

statistical computing; cloud; data parallelism; linear algebra

1 Introduction

"Big data" are growing in volume and diversity at an explosive rate. Data analysis has become increasingly sophisticated and statistical. In many situations, simple database reporting queries are no longer sufficient for deriving insights from high-dimensional and noisy datasets. There are also more people interested in analyzing data than ever before. They are not limited to HPC (High-Performance Computing) centers or big Internet companies; instead, they range from statisticians to biologists, social scientists, and journalists, many of whom have little computing expertise (or access thereto).

The rise of the cloud in recent years presents an interesting opportunity for big-data analytics. There are many successful private clouds (e.g., used by large Internet companies), public cloud service providers (e.g., Amazon's Elastic Compute Cloud, or EC2), as well as software for the cloud (e.g., Apache Hadoop). Users can "rent" a variety of computing options instantaneously from the cloud, pay only for their use according to clear price tags (e.g., Table 1), and avoid hardware acquisition and maintenance costs. Users also benefit from simpler programming models (the most prominent example being MapReduce [7], which Hadoop implements) and a wealth of tools, some of which have started to target machine learning and statistical computing—Apache Mahout, Greenplum/MADlib [5, 10], HAMA [19], RHIPE [9], Ricardo [6], Haloop [3], SystemML [8], ScalOps [1], just to name a few.

Machine type	CPU cores	Memory (GB)	Cost ($/hour)
m1.small	1	1.7	0.065
c1.median	2	1.7	0.165
m1.large	2	7.5	0.260
c1.xlarge	8	7.0	0.660
m1.xlarge	4	15.0	0.520

Table 1: A sample of machine types available from Amazon EC2.

Despite these advantages, it remains frustratingly difficult for most users to use the cloud for any non-trivial statistical analysis of big data—especially when the analysis is novel or ad hoc, and there is no existing program or library already highly tuned for the problem, platform, and budget at hand. First, ***developing*** efficient statistical analysis programs require a great deal of expertise and effort. Popular cloud programming platforms, such as Hadoop, require users to code and think in low-level, platform-specific ways, and, in many cases, resort to extensive manual tuning to achieve acceptable performance. Second, ***deploying*** such programs in the cloud is hard. Users are faced with a maddening array of choices, ranging from hardware provisioning (e.g., type and number of machines to request on Amazon EC2), configuration (e.g., number of map and reduce slots in Hadoop), to implementation alternatives and execution parameters. Some of these choices, as we shall demonstrate through experiments later in this paper, can be critical to meeting deadlines and staying within budget. However, current systems offer little help to users in making such choices.

Our Contributions In this paper we present Cumulon, an end-to-end solution that simplifies both the development and deployment of matrix-based statistical computing programs in the cloud.

When ***developing*** such programs, users of Cumulon will be able to think and code in a natural way using the familiar language of linear algebra. A lot of statistical data analysis (or computationally expensive components thereof) can be succinctly written in the matrix notation and with basic operations such as multiply, add, transpose, etc. Here are two examples that we will revisit later:

- Singular value decomposition (SVD) of a matrix has extensive applications in statistical analysis. In the randomized algorithm

[*]This work is supported by NSF grants 0916027, 0917062, 0964560, and a 2010 HP Labs Innovation Research Award.

SIGMOD'13, June 22–27, 2013, New York, New York, USA.
Copyright 2013 ACM 978-1-4503-2037-5/13/06 ...$15.00.

Algorithm 1: Simplified pseudocode for PLSI. Here, $\circ$ denotes element-wise multiply and $\circ\!\!/$ denotes element-wise divide.

```
Data: $\mathbf{O}$: $m \times n$ sparse word-document matrix, where $m$ is the
      vocabulary size and $n$ is the number of documents;
      $k$: number of latent topics (much less than $m$ and $n$).
Result: $\mathbf{B}$: $n \times k$ dense document-topic matrix;
        $\mathbf{C}$: $m \times k$ dense word-topic matrix;
        $\mathbf{E}$: $k \times k$ diagonal matrix representing topic frequencies.
initialize $\mathbf{B}, \mathbf{C}, \mathbf{E}$;
repeat
    $\mathbf{F} \leftarrow \mathbf{O} \circ\!\!/ (\mathbf{C} \times \mathbf{E} \times \mathbf{B}^\intercal)$;
    $\mathbf{E}' \leftarrow \mathbf{E} \circ (\mathbf{C}^\intercal \times \mathbf{F} \times \mathbf{B})$;
    $\mathbf{C}' \leftarrow (\mathbf{C} \times (\mathbf{I} \circ\!\!/ \mathbf{E}')) \circ (\mathbf{F} \times \mathbf{B} \times \mathbf{E})$;
    $\mathbf{B}' \leftarrow (\mathbf{B} \times (\mathbf{I} \circ\!\!/ \mathbf{E}')) \circ (\mathbf{F}^\intercal \times \mathbf{C} \times \mathbf{E})$;
    $\mathbf{B}, \mathbf{C}, \mathbf{E} \leftarrow \mathbf{B}', \mathbf{C}', \mathbf{E}'$;
until termination condition;
```

of [18] for computing an approximate SVD, the first (and most expensive) step involves a series of matrix multiplies. Specifically, given an $m \times n$ input matrix $\mathbf{A}$, this step uses an $l \times m$ randomly generated matrix $\mathbf{G}$ whose entries are i.i.d. Gaussian random variables of zero mean and unit variance, and computes $\mathbf{G} \times (\mathbf{A} \times \mathbf{A}^\intercal)^k \times \mathbf{A}$. Typically, l is much smaller than m and n, and k is small (say 5). We refer to this step as RSVD-1.

- A popular method in information retrieval and text mining is *Probabilistic Latent Semantic Indexing* (*PLSI*) [13]. The algorithm can be conveniently written in the matrix notation with just a few lines, as shown in Algorithm 1.

Cumulon allows users to rapidly develop in this R-like language, without having to learn MapReduce or SQL, or to worry about how to map data and computation onto specific hardware and platforms.

When ***deploying*** statistical data analysis programs in a cloud, users of Cumulon will be presented with best "deployment plans" meeting their requirements, along with information that is actually useful in making decisions—in terms of completion time and monetary cost. For instance, given a target completion time, Cumulon can suggest the best plan on Amazon EC2 that minimizes the expected monetary cost. This plan encodes choices of not only implementation alternatives and execution parameters, but also cluster provisioning and configuration settings. For example, Figure 1 shows, for various Amazon EC2 machine types under different time constraints, the cost of best deployment plans for RSVD-1, the critical step of the randomized SVD algorithm discussed earlier. Figures like this one show a clear tradeoff between completion time and cost, as well as relative cost-effectiveness of various machine types for the given program, making it easier for users to make informed decisions. Cumulon generates this figure automatically by building cost models and performing cost-based optimization in the space of possible deployment plans. In general, across different machine types and as time constraint varies, the optimal cluster size, configuration, and execution parameters vary; choosing them by hand would have been tedious and difficult.

We make a number of technical contributions in this paper:

1) We show that MapReduce is a poor fit for matrix-based analysis. We adopt a simplified parallel execution model that is more efficient, flexible, and easier for automatic optimization. Instead of building completely new storage and execution engines, we show how to implement Cumulon on top of HDFS and Hadoop, in a way that departs significantly from the MapReduce model. This novel strategy offers large savings over more "traditional" uses of Hadoop for matrix operations, such as SystemML [8].

2) Thanks to Cumulon's focus on matrix-based data analysis, we can model programs not as black boxes, but as plans consisting of a set of commonly used physical operators. Cumulon's simple yet flexible execution model enables us to develop a specialized set of operators with predictable computation and network I/O patterns. We also introduce new operators. For example, "masked matrix multiply" captures an efficient execution strategy involving interacting sparse and dense matrix operations, which has previously only been possible with manual low-level coding.

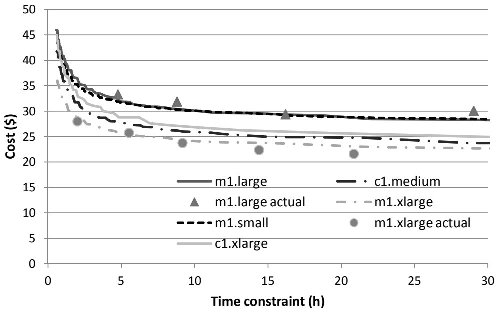

Figure 1: Costs of the optimal deployment plans for RSVD-1 (with $l = 2\text{k}, m = 200\text{k}, n = 200\text{k}, k = 5$) using different machine types under different time constraints. Curves are predicted; costs of actual runs for sample data points are shown for comparison.

3) Building on our understanding of Cumulon's physical operators, we show how to obtain good cost estimates for matrix-based programs in a cloud by benchmarking, simulation, and modeling. Various sources of performance uncertainty in the cloud makes cost estimation particularly challenging. We show how to cope with the variance among the speeds of individual tasks (threads of parallel execution) and account for the behavior of the task scheduler when predicting overall completion time.

4) We demonstrate the benefit of considering a bigger deployment plan space that encodes the choices of not only physical operators and their execution parameters, but also hardware provisioning and configuration settings. Experiments show that, without jointly optimizing these choices, one can get far inferior plans. While Cumulon currently assumes homogeneous clusters, it considers deployment plans that switches clusters dynamically between steps; we give examples where such plans make sense.

5) We devise efficient strategies for searching through this much bigger plan space, by exploiting the properties and common characteristics of our programs for pruning and prioritization. Cumulon can optimize fairly complex programs (such as PLSI, which involves 16 operators) in a reasonable time (under one minute for PLSI) on a desktop with moderate hardware.

6) Hadoop-based Cumulon inherits important features of Hadoop such as failure handling, and is able to leverage the vibrant Hadoop ecosystem. While targeting matrix operations, Cumulon can support programs that also contain traditional, non-matrix Hadoop jobs. For example, we will demonstrate a complete text analytics program consisting of both downloading and preprocessing of documents (to build input matrices) and matrix-based PLSI (Algorithm 1). At the same time, Cumulon is not tied to Hadoop; because of its generic execution model, Cumulon can be extended to multiple storage/execution engines in the future.

2 System Overview and Road Map

We begin with an overview of how Cumulon turns a program into an optimized form ready for deployment in the cloud.

Logical Plan and Rewrites Cumulon first converts the program into a *logical plan*, which consists of *logical operators* corresponding to standard matrix operations, e.g., matrix transpose, add, multiply, element-wise multiply, power, etc. Then, Cumulon conducts a series of rule-based *logical plan rewrites* to obtain "generally bet-

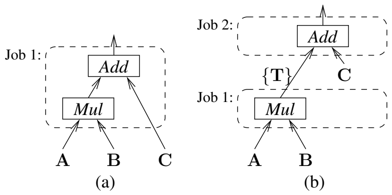

Figure 2: Two physical plan templates for $\mathbf{A} \times \mathbf{B} + \mathbf{C}$. In (b), the output from *Mul*, $\{\mathbf{T}\}$, is a list of matrices whose sum is $\mathbf{A} \times \mathbf{B}$.

ter" logical plans. Such rewrites consider input data characteristics (e.g., matrix sizes and sparsity) and apply linear algebra equivalences to reduce computation or I/O, or increase parallelism.

For example, consider again the expression $\mathbf{G} \times (\mathbf{A} \times \mathbf{A}^\intercal)^k \times \mathbf{A}$ in RSVD-1 from Section 1. The original logical plan raises $\mathbf{A} \times \mathbf{A}^\intercal$ to the k-th power, which repeatedly multiplies large $m \times m$ matrices. However, a better approach (when l and k are small) would be to use associativity of matrix multiply to rewrite the logical plan as:

$$((\cdots((\mathbf{G} \underbrace{\times \mathbf{A}) \times \mathbf{A}^\intercal) \times \cdots \times \mathbf{A}) \times \mathbf{A}^\intercal)}_{2k \text{ multiplies}} \times \mathbf{A},$$

which involves repeatedly multiplying a matrix with only l rows by another matrix, which is much cheaper.

Physical Operators and Plan Templates For each logical plan, Cumulon translates it into one or more *physical plan templates*, each representing a parallel execution strategy for the program. A physical plan template represents a workflow of *jobs*, where each job is in turn a DAG (directed acyclic graph) of *physical operators*. We will provide more details on these concepts in Section 3. For now, think of a job as a unit of work that is executed in a data-parallel fashion, where each thread of execution, or *task*, runs an instance of the physical operator DAG in a pipelined fashion over a different input split. The translation procedure attempts to group operators into as few jobs as possible, but if doing so causes one job to contain multiple resource-hungry physical operators, alternatives will be generated. Also note that one logical operator may be translated into one or more physical operators, and sometimes one physical operator may implement multiple logical operators.[1]

For example, Figure 2 shows two possible physical plan templates for $\mathbf{A} \times \mathbf{B} + \mathbf{C}$. The first one (a) has one job while the second one (b) has two. The reason why two jobs might be preferable will become apparent when we discuss in Section 3 how Cumulon implements matrix multiply in detail. Roughly speaking, the physical operator *Mul* only multiples submatrices from $\mathbf{A}$ and $\mathbf{B}$. In general, to complete $\mathbf{A} \times \mathbf{B}$, these results need to be further grouped and aggregated; in Figure 2b, *Add* performs this aggregation (as well as addition with $\mathbf{C}$) in a separate job. On the other hand, Figure 2a applies in the special case where submatrices multiplied by *Mul* include complete rows from $\mathbf{A}$ and complete columns from $\mathbf{B}$; no further aggregation is needed and *Add* can perform addition with $\mathbf{C}$ in the same job as *Mul*.

Deployment Plan Note that a physical plan template does not completely specify how to execute the program in the cloud. A complete specification, which we call a *deployment plan*, is a tuple $\langle \mathcal{L}, \mathcal{O}, \mathcal{P}, \mathcal{Q} \rangle$, where:

- $\mathcal{L}$ is a physical plan template.
- $\mathcal{O}$ represents parameters settings for all physical operators in $\mathcal{L}$. For example, for the *Mul* operator in Figure 2b, we need to specify the sizes of the submatrices multiplied in each task.
- $\mathcal{P}$ represents hardware provisioning settings, e.g., what machine type and how many machines to reserve on Amazon EC2. In general, such settings may change during the course of execution; for example, when a program shifts from an I/O-intensive stage to a computation-intensive stage, we may want to switch to fewer machines with more powerful CPUs.
- $\mathcal{Q}$ represents configuration settings. For example, for a Hadoop-based implementation of Cumulon, we need to specify the number of map/reduce "slots" per machine. In general, these setting can also change during the course of execution.

Given a physical plan template $\mathcal{L}$ and user requirements (e.g., minimizing monetary cost while capping completion time), Cumulon performs cost-based optimization to find a deployment plan based on $\mathcal{L}$ that satisfies the user requirements. Cumulon then carries out the deployment plan in the cloud. Cumulon will monitor the execution, and alert the user if it detects deviation from the assumptions made or guarantees offered by the optimization.

3 Storage and Execution

As MapReduce has been the prevalent model for data-parallel programming in the cloud, a natural question is whether we can build our support for matrix-based data analysis on this model. Before presenting our approach for Cumulon in Section 3.2, we first show, in Section 3.1, why MapReduce is a poor fit for this type of programs. In Section 3.2, after describing Cumulon's storage and execution models, we sample a couple of interesting physical operators designed for matrices, and show how to implement Cumulon conveniently on top of Hadoop (without adhering to MapReduce).

3.1 Why not MapReduce

A MapReduce job consists of a *map* phase followed by a *reduce* phase. The map phase partitions the input (modeled as key-value pairs) disjointly across multiple map tasks for parallel processing. For the reduce phase, the intermediate output data from the map tasks are grouped by their keys and *shuffled* over the network to be processed by multiple reduce tasks. Each group of intermediate output is processed by the same reduce task to produce final output.

Implementing efficient matrix-based computation within the confines of MapReduce is awkward. In general, in a parallel matrix operation, each task may request specific subsets of input data items. Assignment of input to map tasks in MapReduce can be controlled to some extent by representing input in different ways (such that all data needed by a task is packed as one key-value pair), or with custom input splitting supported by some implementations of MapReduce (e.g., Hadoop). However, such control is still limited in supporting the general data access patterns of matrix operations.

In particular, map tasks in MapReduce are supposed to receive *disjoint* partitions of input, but for many matrix operations, one input data item may be needed by multiple tasks. A workaround in MapReduce is for mappers to read in the input items, replicate them—one copy for each reduce task that needs it, and then shuffle them to reduce tasks. The map phase thus does nothing but to make extra copies of data, which could have been avoided completely if map tasks are allowed to receive overlapping subsets of input.

Next, we illustrate the limitation of MapReduce using a concrete example of matrix multiply, and discuss the inefficiency in existing MapReduce-based implementations.

Example: Matrix Multiply Consider multiplying two big matrices $\mathbf{A}$ (of size $m \times l$) and $\mathbf{B}$ (of size $l \times n$). Let $\mathbf{C}$ denote the result (of size $m \times n$). On a high level, virtually all efficient parallel matrix multiply algorithms work by dividing input into submatrices: $\mathbf{A}$ into $f_m \times f_l$ submatrices, each with size ${}^{m}\!/_{f_m} \times {}^{l}\!/_{f_l}$, and

[1] We will not discuss logical plan rewrites and translation into physical plan templates further: because of space constraints, they are not the focus of this paper. Their solutions are orthogonal from and should readily work with the techniques in this paper.

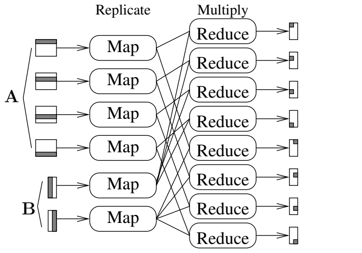

Figure 3: MapReduce job for $\mathbf{A} \times \mathbf{B}$ using SystemML's RMM; $f_l = 1$ is required.

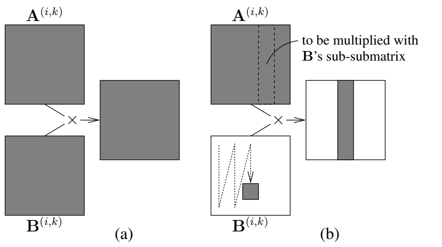

Figure 4: Options for multiplying submatrices within a task. Shaded portions are kept in memory.

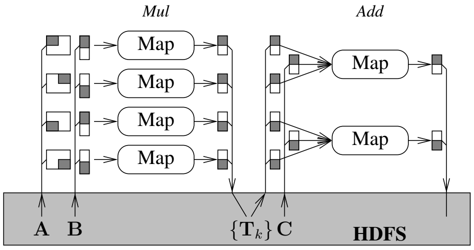

Figure 5: Cumulon's map-only Hadoop jobs for $\mathbf{A} \times \mathbf{B} + \mathbf{C}$ using the physical plan template of Figure 2b.

$\mathbf{B}$ into $f_l \times f_n$ submatrices, each with size ${}^{l}/_{f_l} \times {}^{n}/_{f_n}$. We call f_m, f_l, and f_n *split factors*. The output will consist of $f_m \times f_n$ submatrices. Let $\mathbf{M}^{(i,j)}$ denote the submatrix of $\mathbf{M}$ at position (i, j). The output submatrix at position (i, j) is then computed as $\mathbf{C}^{(i,j)} = \sum_{k=1}^{f_l} \mathbf{A}^{(i,k)} \times \mathbf{B}^{(k,j)}$. Computation typically proceeds in two steps: first, pairs of submatrices from $\mathbf{A}$ and $\mathbf{B}$ are multiplied in parallel; second, the results of the multiplies are grouped (by the output submatrix position they contribute to) and summed. Note that if $f_l = 1$, the second step is not needed.

Indeed, matrix multiply is an example where one input submatrix may be needed by multiple tasks in the first step (unless $f_m = f_n = 1$, a special case that we will discuss later). If we adhere to MapReduce, we have to use the map phase to replicate the input submatrices and shuffle them to the reduce phase to be multiplied. Next, if $f_l \neq 1$, we need a second MapReduce job to perform the required summation, adding even more overhead. SystemML [8] takes this approach (called *RMM* there), but to avoid the second MapReduce job, it requires $f_l = 1$ (see Figure 3 for illustration).

However, SystemML's RMM turns out to be suboptimal on many accounts. First, the map phase performs no useful computation; input replication and shuffling are just expensive workarounds to allow reduce tasks to access data they need. Second, $f_l = 1$ severely limits the choice of execution strategies: it basically amounts to multiplying rows of $\mathbf{A}$ with columns of $\mathbf{B}$. It is well known in the literature on out-of-core linear algebra [21] that such an aspect ratio is suboptimal for matrix multiply (assuming sufficiently large m, l, and n). The number of element multiplications per task in this case is only linear in the number of input elements per task; in contrast, using square submatrices of size $k \times k$, for example, would yield k^3 multiplications per $2k^2$ input elements. Since the total number of element multiplications across all tasks is fixed ($m \times l \times n$), strategies that require fewer input elements for the same number of multiplications have lower I/O. We will experimentally confirm the suboptimality of SystemML's RMM strategy in Section 6.1.

In the other special case of $f_m = f_n = 1$, we basically multiply columns of $\mathbf{A}$ with rows of $\mathbf{B}$: each submatrix multiply generates a matrix of the same size as the final output $\mathbf{C}$, and these matrices simply need to be added to produce $\mathbf{C}$. Note that in this case the submatrix multiply tasks work on disjoint input data, though each task needs to draw a specific pair of submatrices from $\mathbf{A}$ and $\mathbf{B}$. If we relax the MapReduce model to allow custom input splitting, it would be possible to use a map phase to perform all submatrix multiplies, and a reduce phase to perform the final add. However, under a "pure" MapReduce model, we would need one full MapReduce job for the submatrix multiplies and another one for the add: the map phase in both jobs serves the sole purpose of routing appropriate data to reduce tasks, which still incurs considerable overhead. This two-job strategy for $f_m = f_n = 1$ is called *CPMM* in SystemML [8]. RMM and CPMM together are the two matrix multiply strategies supported by SystemML.

HAMA [19] supports arbitrary split factors for matrix multiply. It assumes that input has already been preprocessed for a MapReduce job such that each map task receives the two submatrices to be multiplied as one input data item. The result submatrices are then shuffled to reduce tasks to be added. Thus, HAMA carries out all computation in a matrix multiply in a single MapReduce job. However, the preprocessing step would still take another MapReduce job to replicate the input submatrices.

3.2 The Cumulon Approach

Having seen how the pure MapReduce model is not suitable for matrix operations, we now turn to a simpler but more flexible model.

Storage Since Cumulon targets matrix-based computation, it provides an abstraction for distributed storage of matrices. Matrices are stored and accessed by *tiles*. A tile is a submatrix of fixed (but configurable) dimension. Cumulon's execution model guarantees that, at any given time, a tile has either multiple concurrent readers, or one single writer.

Tile size is chosen to be large enough to amortize the overhead of storage and access, and to enable effective compression. However, they should not be so big that they limit execution options (e.g., split factors in matrix multiply). See Section 6 for the default setting in experiments. Within a tile, elements are stored in column-major order. Both sparse and dense formats are supported: the sparse format enumerates indices and values of non-zero elements, while the dense format simply stores all values.

Jobs and Tasks A Cumulon program executes as a workflow of *jobs*. A job reads a number of input matrices and writes a number of output matrices; input and output matrices must be disjoint. Dependencies among jobs are implied by dependent accesses to the same matrices. Dependent jobs execute in serial order. Each job executes as multiple independent *tasks* that do not communicate with each other. All tasks within the job run identical code, but read different (possibly overlapping) parts of the input matrices (*input splits*), and write disjoint parts of output matrices (*output splits*). Hence, this execution model can be seen as a restricted variant of the *Bulk Synchronous Parallel* model [22] where data are only communicated through the global, distributed storage in the unit of tiles.

It is worth noting that tiles do not correspond to input/output splits. A task may work on input/output splits that each consist of multiple tiles. Also, unlike MapReduce, input splits do not need to be disjoint across tasks, and a task can read its input split and write its output split anytime during its execution.

Slots and Waves Available hardware is configured into a number of *slots*, each of which can be used to execute one task at a time. A *scheduler* assigns tasks to slots. If a job consists of more tasks than slots, it may take multiple *waves* to finish. Each wave consists of a number of concurrently executing tasks no more than the number of slots. There may not be a clear boundary between waves, as a slot can be assigned another task in the same job immediately after the current task completes. However, as stated earlier, dependent jobs do not start until the current job is completely done.

Physical Plan Templates and Operators As defined in Section 2, a *deployment plan* completely specifies how to execute a Cumulon program. The *physical plan template* component ($\mathcal{L}$) of the deployment plan specifies a workflow of jobs, where each job is a DAG of *physical operators*. Each physical operator has a list of parameters, whose settings (by the $\mathcal{O}$ component in the deployment plan) control their execution behavior. During job execution, each task executes an instance of the DAG. Similar to database query execution, the physical operators execute in a pipelined fashion through an iterator-based interface. To reduce the overhead of element-wise data passing, the unit of data passing among iterators is a tile (when appropriate). Physical operators can read and write tiles in the input and output splits (respectively) assigned to their tasks.

Example Recall the two physical plan templates for $\mathbf{A} \times \mathbf{B} + \mathbf{C}$ in Figure 2. Here we fill in more details. The *Mul* physical operator has split factors f_m, f_l, and f_n as its parameters (they are not the only ones; more discussion will follow shortly): set by $\mathcal{O}$, they control the dimensions of two submatrices being multiplied. *Mul* reads these submatrices directly from distributed storage.

The physical plan template in Figure 2a works for $f_l = 1$, where *Mul* produces a submatrix of $\mathbf{A} \times \mathbf{B}$ directly (no summation is required), and passes this submatrix to *Add* in the same job (one tile at a time) to be added to the corresponding submatrix of $\mathbf{C}$.

The physical plan template in Figure 2b works in the general case of $f_l > 1$ and requires two jobs. The first job multiplies submatrices from $\mathbf{A}$ and $\mathbf{B}$, and writes f_l intermediate matrices, where the k-th intermediate matrix $\mathbf{T}_k$ consists of submatrics $\mathbf{T}_k^{(i,j)} = \mathbf{A}^{(i,k)} \times \mathbf{B}^{(k,j)}$; each instance of *Mul* is responsible for one submatrix in one of the intermediate matrices. The second job computes $(\sum_{k=1}^{f_l} \mathbf{T}_k) + \mathbf{C}$; each instance of *Add* reads corresponding tiles from $\mathbf{T}_1, \ldots, \mathbf{T}_k, \mathbf{C}$, adds them, and writes an output tile.

A Closer Look at *Mul* So far, our discussion of *Mul* has remained at a high level; next, we discuss how two input submatrices $\mathbf{A}^{(i,k)}$ and $\mathbf{B}^{(k,j)}$ are actually multiplied. There are many options, with different performance tradeoffs. We begin with two options illustrated in Figure 4. **1)** The most straightforward option is to read two submatrices entirely into memory and multiply them using a highly tuned BLAS library. This option has good CPU utilization but also high memory requirement. The amount of memory available to each slot limits the size of the submatrices that can be multiplied without thrashing. **2)** Another option, aimed at reducing memory requirement, is to read one submatrix in memory and stream in the other one in a specific order tile by tile. Say $\mathbf{A}^{(i,k)}$ is read and buffered in memory. Conceptually, each column of output tiles is produced by multiplying $\mathbf{A}^{(i,k)}$ with each column of tiles in $\mathbf{B}^{(k,j)}$. To this end, we reserve memory for a column of output tiles, and stream in tiles of $\mathbf{B}^{(k,j)}$ in column-major order. The p-th tile in the current column of $\mathbf{B}^{(k,j)}$ would be multiplied with the p-th column of tiles in $\mathbf{A}^{(i,k)}$ and accumulated into the column of output tiles. While this option has a lower memory requirement, its multiplications are performed in a more piecemeal fashion, so it tends to result in lower CPU utilization than the first option.

There actually exist a range of options between the two above. Besides the split factors f_m, f_l, f_n, which control the dimensions of submatrices to be multiplied, *Mul* also includes parameters that further specify the input submatrix to buffer in memory and the *granularity of streaming* the other submatrix as "sub-submatrices." These parameters provide a smooth tradeoff between CPU utilization and memory requirement. The first option above is the case where $\mathbf{B}^{(k,j)}$ has just one sub-submatrix; the second option is where the sub-submatrices are tiles. Further details are omitted.

MaskMul Besides improving standard physical operators (e.g., making *Mul* more flexible and allowing *Add* to take multiple inputs), we have also found the need to introduce new operators in order to capture efficient execution strategies for complex expressions. *MaskMul* is one such example.

To motivate, consider Line 3 of PLSI (Algorithm 1), where a large, but very sparse, matrix $\mathbf{O}$ is element-wise divided by the equally large result of a chain of dense matrix multiplies. With standard matrix operators, we will be forced to fully evaluate the expensive dense matrix multiplies. A well-known trick in hand-coded PLSI implementations is to exploit the sparsity in $\mathbf{O}$ to avoid full dense matrix multiplies: if a particular element of $\mathbf{O}$ is 0, there is no need to compute the corresponding element in the result of multiplies, because element-wise divide would have returned 0 anyway.

Therefore, we introduce a new physical operator, *MaskMul* (for *masked* matrix multiply). In addition to input matrices $\mathbf{A}$ and $\mathbf{B}$ to be multiplied, *MaskMul* receives a "mask" in the form of a sparse matrix $\mathbf{M}$ to be element-wise multiplied or divided by $\mathbf{A} \times \mathbf{B}$. Execution of *MaskMul* is driven by the non-zero elements in $\mathbf{M}$. Conceptually, we multiply row i of $\mathbf{A}$ with column j of $\mathbf{B}$ only if $m_{i,j} \neq 0$. Details are omitted because of space constraints.[2]

As we will see in Section 6.1, *MaskMul* can improve performance dramatically. In particular, it speeds up Line 3 of PLSI (Algorithm 1) by more than an order of magnitude (Figure 11).

Need for Automatic Optimization From the discussion of physical operators, it is clear that we want automatic optimization of matrix-based data analysis programs. Even for simple programs with the most basic matrix operations, there are many choices to be made. For example, for Figure 2, should we choose two jobs, or one job with potentially suboptimal split factors? For matrix multiply, what are the best choices of split factors and streaming granularity? Depending on sparsity, should we use *Mul* or *MaskMul*? Some of these choices are related to each other, and to hardware provisioning and configuration settings that affect number of tasks (and split size per task), memory, CPU utilization, relative costs of CPU and I/O, etc. Making these choices manually may be infeasible, error-prone, or simply not cost-effective.

Implementation in Hadoop Our discussion so far only assumes generic storage and execution platforms. While this generality is by design—as we want Cumulon to be able to work with a variety of alternative platforms in the future—our current implementation of Cumulon is based on Hadoop and HDFS, which allows us to readily leverage its features, flexibility, and ecosystem of tools and users. We now describe this implementation, highlighting how it departs from the traditional MapReduce-based usage of Hadoop.

We use HDFS for distributed matrix storage. A matrix is stored using one or more data files in HDFS. Tiles of a matrix written by the same task go into the same data file. For each matrix, we also store metadata (including information such as dimension and sparsity) and index (which maps the position of a tile to a data file and the offset and length therein) in HDFS. Additional details are omitted because of space constraints.

Each Cumulon job is implemented as a *map-only* Hadoop job. Cumulon slots correspond to Hadoop map slots (we do not have reduce slots because we never use the reduce phase). Unlike MapRe-

[2] Although the optimization enabled by *MaskMul* is reminiscent of "pushing selections down through a join" in databases, it cannot be achieved in our context using just rewrite rules involving standard matrix operators. The reason is that the mask cannot be pushed below the multiply to its input matrices—unless the mask has rows or columns consisting entirely of zeros, every element of the input matrices will be needed by some non-zero element of the mask. Thus, we need the new operator *MaskMul* to handle this non-standard filtering of computation.

duce, a map task does not receive input as key-value pairs; instead, it simply gets specifications of the input splits it is responsible for, and reads directly from HDFS as needed. In-memory dense matrix multiplies call a (multi-threaded) JBLAS library. Each map task also writes all its output to HDFS; there is no shuffling.

For example, Figure 5 illustrates how Cumulon implements the physical plan template in Figure 2b with Hadoop and HDFS. Contrasting with SystemML's use of Hadoop in Figure 3, we see that Cumulon is simpler, more flexible (e.g., by supporting any choice of split factors), and has less overhead (by avoiding map tasks that do nothing but route and replicate data). These advantages will be confirmed by experiments in Section 6. Section 6.1 will also compare with another possible implementation of Cumulon on Hadoop that use reduce tasks (to get fewer jobs); we will see that the map-only approach described here works better.

4 Cost-Based Optimization

Overview We now turn to cost-based optimization of deployment plans for matrix-based data analysis in the cloud. Given a physical plan template $\mathcal{L}$—recall from Section 2 that Cumulon obtains possible $\mathcal{L}$'s from the original logical plan through rewrites and translation—we would like to find a deployment plan for $\mathcal{L}$ that is optimal in terms of completion time and total (monetary) cost. Recall that a deployment plan for $\mathcal{L}$ is a tuple $\langle \mathcal{L}, \mathcal{O}, \mathcal{P}, \mathcal{Q} \rangle$, where $\mathcal{O}$ represents the physical operator parameter settings, $\mathcal{P}$ represents the hardware provisioning settings, and $\mathcal{Q}$ represents the configuration settings. Let $\mathfrak{T}(\cdot)$ and $\mathfrak{C}(\cdot)$ denote the completion time and cost of a deployment plan, respectively. In this paper, we focus on:

$$\textbf{minimize}_{\mathcal{O},\mathcal{P},\mathcal{Q}}\ \mathfrak{C}(\langle \mathcal{L}, \mathcal{O}, \mathcal{P}, \mathcal{Q} \rangle)\ \textbf{s.t.}\ \mathfrak{T}(\langle \mathcal{L}, \mathcal{O}, \mathcal{P}, \mathcal{Q} \rangle) \leq T_{\text{max}};$$

i.e., minimizing cost given a completion time constraint (T_{max}). Other problem formulations may also be appropriate (e.g., minimizing time given a cost budget, or finding all Pareto-optimal plans); techniques in this section can be extended to these formulations.

We currently make two simplifying assumptions. First, while we support intra-job parallelism (i.e., we execute each job using parallel tasks), we do not consider inter-job parallelism (i.e., two jobs cannot execute at the same time even if they are not dependent on each other). Second, at any point during the execution of a job, the cluster is homogeneous (i.e., consisting of identical machines). What partially compensates for these limitations is our support for dynamic cluster switching between jobs. Thus, the cluster can be heterogeneous across jobs and over time.

Hadoop and Amazon EC2 Currently, Cumulon is implemented on top of Hadoop and targets deployment on Amazon EC2. Under this setting, we can spell out the components $\mathcal{P}$ and $\mathcal{Q}$ of a deployment plan. $\mathcal{P}$ specifies, for each job, the type and number (N) of machines to reserve on EC2; a change in either type or number of machines across adjacent jobs implies cluster switching. $\mathcal{Q}$ specifies the Hadoop/HDFS configuration settings for each job. From our experience, the most relevant setting with a significant impact on plan performance is the number of map slots (S) per machine; we will therefore focus on this setting. Appropriate values for other settings can either be derived using S (such as memory available for each task) or chosen in ways independent from particular plans (such as HDFS block size).

For EC2, the monetary cost of a cluster is calculated by multiplying the hourly per-machine rate for the cluster type, the number of machines in the cluster, and the number of hours (rounded up to the nearest integer) that the cluster is in use. In this paper, we simplify the cost function by assuming no rounding up, as we estimate expected costs anyway. It is straightforward to extend our techniques to work with rounding and other billing schemes.

Next, we discuss how to estimate the completion time and cost of a deployment plan (Section 4.1), and how to search the space of feasible plans for the optimal one (Section 4.2). Although much of the discussion will be specific to Hadoop and EC2, our technique should be applicable to other scenarios as well.

4.1 Time and Cost Estimation

On EC2, the total completion time and cost of a deployment plan come from the following sources: **1)** running Hadoop jobs; **2)** cluster initialization; **3)** cluster switching; **4)** data ingress/egress. (1) is the most significant source and will be the emphasis of this section. Time and cost for (2), (3), and (4) are relatively easier to estimate. (2) and (3) depend on the size and type of the clusters involved. (3) and (4) depend the amount of data transferred. For (3), the matrices that need to be copied to the new cluster can be obtained via live variable analysis. For (4), the cost depends on the method used for data ingress/egress. Currently we assume that all input/output data of a program reside on Amazon EBS, but other methods and cost functions can be plugged in. For (2), (3), and (4), we will not go into additional details because of space constraints.

Since the cost of running a Hadoop job on EC2 can be readily calculated from the completion time, we will focus on estimating the expected completion time of a job in this section. Our overall approach is to build a model for job completion time using data collected from benchmarking and simulation. Since the number of possible settings of $\mathcal{L}$, $\mathcal{O}$, $\mathcal{P}$, and $\mathcal{Q}$ is huge, we cannot afford a single high-dimensional model; instead, we decompose it into components and build each component separately. In the remainder of subsection, we first discuss how to estimate completion time of individual tasks (Section 4.1.1) by modeling computation and network I/O separately, and then how to estimate job completion time from task completion time (Section 4.1.2).

Several challenges differentiate cost estimation in Cumulon from non-cloud, database settings. First, cardinality estimation, key in databases, is a not an issue in our setting because the dimensions of matrices (even intermediate ones) are easy to derive at compile time. Instead, the closest analogy in our setting is estimation of matrix sparsity, which affects the cost of sparse matrix operations. Second, in matrix-based computation, CPU accounts for a significant fraction of the total running time, so unlike the database setting, we cannot count only I/O (in our case, network I/O). Third, performance uncertainty that arises from parallel execution in the cloud makes estimation especially challenging. For example, we need to model how variance among individual task completion times affects the overall job completion time (even if we care only about its average). Finally, we note that accurate cost modeling of arbitrary statistical computing workloads is extremely difficult, because it entails predicting data- and analysis-dependent convergence properties. Techniques in this section aim at predicting "per-iteration" time and cost instead of the number of iterations; Section 5 discusses how Cumulon handles this challenge pragmatically.

4.1.1 Estimating Task Completion Time

We estimate the running time of a task as the sum of three components: computation, network I/O (from/to the distributed matrix store), and task initialization. The simplicity of this approach stems from the simplicity of the Cumulon execution model, which gives map tasks direct control of their I/O and eliminates the reduce phase and shuffling. Task initialization is a fixed overhead given a slot and is easy to estimate; we focus on computation and network I/O next.

Estimating Computation Time of a Task Recall that a task consists of a DAG of physical operators. We build a model of computation time for each operator and compose (add) them into a model for the task. A possible approach is to analytically derive a FLOP

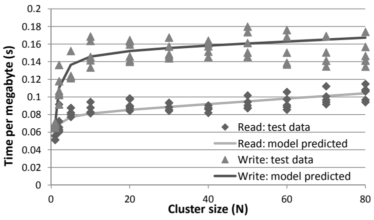

Figure 6: Estimated vs. actual times per unit I/O for machine type c1.medium with $S = 2$. The program measured here multiplies two dense matrices.

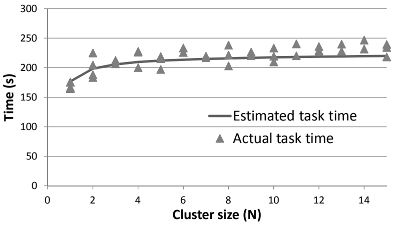

Figure 7: Estimated vs. actual average task completion times for machine type c1.medium with $S = 2$. The program here multiplies two 6k × 6k dense matrices.

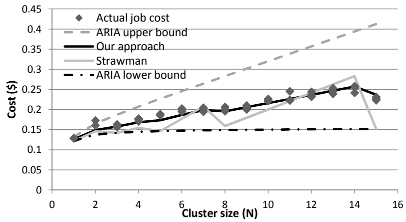

Figure 8: Actual job costs vs. predictions from the strawman approach, ours, as well as ARIA upper/lower bounds. The job multiplies two dense matrices of sizes 12k×18k and 18k × 30k. Machine type is c1.medium; $q = 30$; $S = 2$.

(floating-point operations) count model for each operator, but we have found that FLOP counts alone do not lead to an accurate estimate of computation time. For example, a JBLAS implementation of matrix multiply is easily orders-of-magnitude faster than hand-coded loops, despite the same FLOP counts; a sparse matrix multiply is much slower than a dense one with the same FLOP count. Therefore, we build a model for each operator by benchmarking.

In general, the model inputs include machine type, number of slots per machine, operator parameter settings, input split size, and input sparsity. There is no need to consider the number of machines in the cluster because computation is local. For some operators, the model can be pared down further. For example, for dense matrix *Mul*, we only need to benchmark for combinations of machine type, number of slots, and dimensions of sub-submatrices it chooses to multiply-accumulate each time using JBLAS. Then, the computation time of the whole *Mul* operator can be computed by multiplying the estimated time per JBLAS call by the number of such calls, which can be easily and analytically derived from the input split size, split factors, and granularity of streaming (recall Section 3.2).

Instead of benchmarking every possible setting of model inputs, we sample the space of feasible settings. For example, for *Mul*, machine type and number of slots limit the amount of memory available to each slot, and in turn bounds the sizes of sub-submatrices, which we additionally constrain to be integral multiples of tile sizes. The number of slots is also naturally limited on a given machine type; it makes no sense to allot a large number of slots on a machine with few cores because matrix computation is CPU-intensive. We repeat the benchmark for each setting multiple times; beside the mean, the sample variance will also be useful later (Section 4.1.2). Estimates for unseen settings are obtained using interpolation from neighboring benchmarked settings.

Though this approach is simple, we have found it very effective so far, both for highly predictable dense matrix operators, and for sparse matrix operators where the distribution of non-zero entries is random enough with respect to how input is split—this case happens, for example, to our PLSI program because when it generates $\mathbf{O}$, it orders documents and words randomly (by their hashes). Future work is needed to better model the impact of skew in sparsity; new models can be readily incorporated into Cumulon.

Estimating Network I/O Time per Task Modeling of network I/O time is quite different from modeling computation. First, the average I/O speed depends on the size of the cluster (see Figure 6 for an example). Second, the total I/O volume is a reasonable predictor for I/O time (unlike FLOP count for computation time). Therefore, we estimate the total I/O volume per task, model time per unit I/O, and multiply them to obtain the I/O time estimate.

The total I/O volume per task is the sum of the I/O volume for each of its operators, which can be analytically derived from input split size, input sparsity, and relevant operator parameters settings.

To model the time per unit I/O, or inverse speed v^{-1}, we note that it depends on the machine type, number of slots per machine (S), and number of machines (N). Reads and writes need to be modeled differently. Because of space constraints, we focus on reads here. In our setting, some reads may be local because some HDFS blocks (or replicas thereof) may happen to reside on machines reading them. For each machine type and S, we benchmark local and remote reads. Let $v_{\text{r.local}}^{-1}$ denote the inverse speed (seconds per MB transferred) of local reads, and let $v_{\text{r.remote}}^{-1}$ denote the inverse speed of remote reads. Given a machine type and S, we estimate $v_{\text{r.local}}^{-1}$ as a constant and $v_{\text{r.remote}}^{-1}$ an affine function of N from the benchmark results. (It turns out that a larger N slows down $v_{\text{r.remote}}^{-1}$, but the effect is small.) We then estimate the inverse read speed v_{r}^{-1} as $\frac{1}{N}(\gamma v_{\text{r.local}}^{-1} + (N-\gamma) v_{\text{r.remote}}^{-1})$, where γ is the HDFS replication factor. Intuitively, $\gamma/_N$ is the probability that a read is local. Again, just like for computation time, we also derive a variance estimate for v^{-1} from sample variances obtained from benchmarking.

Figure 6 compares our model predictions with actual times per unit of data read/written by a dense matrix multiply program. A total of 108 actual measurements are plotted (60 of which are from multiplying two 6k × 6k matrices, and 48 are from multiplying a 4k × 8k matrix with an 8k × 8k one). The read/write models are each trained using 22 data points (disjoint from the 108 above) obtained by benchmarking the multiplication of 6k×6k matrices. The figure shows that our models are quite effective, with coefficients of determination 0.92 (for reads) and 0.91 (for writes).

Putting It Together To estimate the expected completion time of a task, we add our estimates for the three components: computation, I/O, and task initialization. Figure 7 shows our estimate to be fairly consistent with the actual average task completion time for dense matrix multiply, over a wide range of cluster sizes. In this figure, each of the 40 data points plotted represents the measured average task completion for a test run (not used for training models). To obtain the variance in task completion time (which will be needed in Section 4.1.2 for estimating expected job completion time), we assume that the three component times are uncorrelated, and simply add their variances. The variance for the I/O time component is derived by scaling the standard deviation (root of variance) in v^{-1} (in seconds per MB) by the I/O volume (in MB).

4.1.2 Estimating Job Completion Time

Before presenting our approach to estimating job completion time, we start by discussing a strawman approach and a previous approach by *ARIA* [23]. Let q denote the number of tasks in a job (which can be derived from the input split size and total input size), and let μ denote the mean task completion time. Note that the total number of slots available for executing these tasks is NS.

Strawman With the strawman approach, we simply compute the number of waves as $\lceil q/_{NS} \rceil$, and multiply it by μ to get the expected job completion time. If this approach works, we should see, as depicted by the plot for strawman in Figure 8, lots of jaggedness in the completion times (and hence costs) of a given job when N

varies. Here, since each task remains the same, its completion time changes little with N. As N increases, $\lceil ^q/_{NS} \rceil$ does not decrease smoothly; it decreases only at specific "breakpoints" of N. Thus, by this model, job completion time dips only at these points and remains flat otherwise; cost dips at the same points but goes up (because we get charged by N) otherwise. However, the actual job completion time/cost behavior, also shown in Figure 8, is smoother.

This discrepancy can be explained by the variance in task completion times and the behavior of the task scheduler. Suppose that the last wave has just a few remaining tasks. If every task takes exactly μ time, the job completion time will indeed be μ times the number of waves. However, because the variance in task completions, some slots will finish earlier, and the scheduler will assign the remaining tasks to them; thus, the last wave may complete under μ. Overall, the variance in task completion times actually make the job completion time behavior smoother over N.

ARIA ARIA [23] characterizes task completion times by the observed average, minimum, and maximum among past runs of the same exact task. From these quantities, ARIA analytically derives an lower bound and an upper bound for job completion time. Figure 8 shows how these bounds translate to bounds on job cost (here ARIA is given 100 observations). ARIA estimates the expected job time/cost to be the average of the lower/upper bounds. From Figure 8, we see that the bounds are quite loose; although their average matches the general trend in actual job costs, it is too smooth and fails to capture the variations—its root-mean-square error (relative to actual average job costs) is 0.0189 (compared with 0.0078 for our approach). Also, ARIA's analytical bounds do not consider different custom scheduler behaviors, which are possible in Hadoop.

Our Approach To account for the effect of variance (denoted σ^2) in task completion times, Cumulon models job completion time as a function of q, NS, and μ as well as σ^2. Specifically, the function has form $\mu(\lceil ^q/_{NS} \rceil + (\frac{q \mod NS}{NS} - \alpha(\frac{\sigma}{\mu})) \cdot \beta(\frac{\sigma}{\mu}))$. Intuitively, the number of waves is adjusted by a fraction that accounts for σ^2 and the scheduler behavior. We take a simulation-based approach to support custom scheduler behavior. Given q, NS, and σ/μ, we draw task completion times from a truncated Gaussian with $(1, (\sigma/\mu)^2)$, and simulate the scheduler to obtain the expected job time. Using simulation results, we fit $\alpha(\frac{\sigma}{\mu})$ and $\beta(\frac{\sigma}{\mu})$ as low-degree polynomials. As Figure 8 shows, this approach models the job costs best, capturing both general trends and local variations.

To summarize, starting with models for components of the individual task completion times, we have shown, step by step, how to accurately estimate expected costs of jobs. Capturing the variance in task completion times is crucial in our approach. In this paper we have focused on predicting expected job costs; further quantifying the error variance would be a natural next step.

4.2 Search Strategy

Having seen how to estimate expected completion time and cost of a deployment plan, we now discuss how to solve the optimization problem posed at the beginning of this section:

$$\textbf{minimize}_{\mathcal{O},\mathcal{P},\mathcal{Q}}\ \mathfrak{C}(\langle \mathcal{L}, \mathcal{O}, \mathcal{P}, \mathcal{Q} \rangle)\ \textbf{s.t.}\ \mathfrak{T}(\langle \mathcal{L}, \mathcal{O}, \mathcal{P}, \mathcal{Q} \rangle) \leq T_{\text{max}}.$$

As Section 6.2 will demonstrate with experiments, it is necessary to jointly consider operator parameter settings as well as hardware provisioning and configuration settings. The main challenge addressed in this section is how to cope with the large search space.

Establishing the Range of Cluster Sizes The number of machines (N) is a parameter in $\mathcal{P}$ with a potentially large range. We can narrow down this range by making two observations. For a given job running on a cluster with a particular machine type: **1)** as we decrease N, the job's best possible expected completion time cannot decrease; **2)** we can find a function $\check{c}(N)$ that lower-bounds the expected cost of the job on a cluster of size N, such that $\check{c}(N)$ is non-decreasing in N. Both observations are intuitive. (1) states that a job cannot run slower with more resources. While not always true for extremely large values of N (where the overhead of distribution could make a job run slower), the observation will likely hold in the range of N we actually consider in practice (unless we are given unreasonably tight T_{max}). As for (2), while cost may not always increase with N, the general trend is increasing (as exemplified by Figure 8) as the overhead of parallelization kicks in. Our model of job time in Section 4.1.2 allows us to derive $\check{c}(N + 1)$ after searching the deployment plans for a cluster of size N (details omitted because of space constraints).

Given an upper bound on completion time, (1) naturally leads to a lower bound $\check{N}$ on N for meeting the completion time constraint. Using a procedure $\texttt{opt}_{\mathcal{O},\mathcal{Q}}$ (see Algorithm 2 for pseudocode) that finds the optimal deployment plan given $\mathcal{L}$ and $\mathcal{P}$, Cumulon finds lower bound $\check{N}$ efficiently by performing an exponential search in N, using $O(\log \check{N})$ calls to $\texttt{opt}_{\mathcal{O},\mathcal{Q}}$.

Given an upper bound on cost, (2) naturally leads to an upper bound on N required for beating the given cost. Starting from the lower bound on N found earlier, we can examine each successive N, which involves calling $\texttt{opt}_{\mathcal{O},\mathcal{Q}}$ with this N, until $\check{c}(N + 1)$ (obtained as a by-product of the $\texttt{opt}_{\mathcal{O},\mathcal{Q}}$ call) exceeds the given cost.

Searching for Plans Without Cluster Switching We are now ready to describe the algorithm $\texttt{opt}_{\mathcal{O},\mathcal{P},\mathcal{Q}}$ (Algorithm 2) for finding the optimal deployment plan for $\mathcal{L}$ without cluster switching. The idea is simple. For each machine type, we establish the range of relevant cluster sizes as described earlier—the time constraint is given; the cost constraint is given by the best plan found so far, because any plan with a higher cost need not be considered.

The innermost loop of $\texttt{opt}_{\mathcal{O},\mathcal{P},\mathcal{Q}}$ calls $\texttt{opt}_{\mathcal{O},\mathcal{Q}}$. Since $\texttt{opt}_{\mathcal{O},\mathcal{Q}}$ is given a cluster of fixed type and size, $\texttt{opt}_{\mathcal{O},\mathcal{Q}}$ can optimize each job $j \in \mathcal{L}$ independently. Currently, we use a straightforward method to find the optimal $\mathcal{O}$ and $\mathcal{Q}$ for each j. We loop through possible values of S (which are naturally capped by a small multiple of the number of cores per machine, because our programs are CPU-intensive), and, for each S, sample the space of feasible operator parameter settings such that the total memory required by j's operators does not exceed a slot's available memory. As mentioned earlier, we also extend this method to compute a cost lower bound for a cluster of the same type and size $N + 1$ (details omitted). Finally, $\texttt{opt}_{\mathcal{O},\mathcal{Q}}$ returns the optimal $\mathcal{O}$ and $\mathcal{Q}$ for $\mathcal{L}$, which consists of the optimal settings for each job in $\mathcal{L}$. Although simple, this method is fast enough in practice, because each job tends not to have many operators (see Section 6.2 for some optimization time numbers). The modular design of Cumulon's optimization algorithm allows more sophisticated methods to be plugged in if needed.

Searching for Plans with Cluster Switching Cluster switching blows up the already big search space by another factor exponential in the number of jobs in the program. Therefore, we have to resort to heuristics. We conjecture that the for real programs, there likely exist near-optimal deployment plans that do not switch clusters many times. One reason is that switching incurs considerable overhead. Another reason is that real programs often contain a few steps whose costs are dominating; it might make sense to switch clusters for these steps, but additional switching between other steps will unlikely result in significant overall improvement.

Operating under this assumption, we search the plan space by first considering plans with no switches (using $\texttt{opt}_{\mathcal{O},\mathcal{P},\mathcal{Q}}$ described above), and then those with one more switch in each successive iteration. This prioritization allows the search to be terminated when optimization runs out of time, and ensures that the most promising plans are considered first.

Algorithm 2: Optimizing deployment plan with no cluster switching.

```
opt_{O,Q}(L, P): L: physical plan template;
                 P: type and number of machines in the cluster.
returns: deployment plan with lowest cost for given L and P;
  if two return values are expected, the second lower-bounds the cost
  on larger clusters of the same type (details omitted).
$\mathcal{O}, \mathcal{Q} \leftarrow \varnothing, \varnothing$;
foreach job $j \in \mathcal{L}$ do
    $C_{\text{best}}, S_{\text{best}}, O_{\text{best}} = \infty, \perp, \perp$;
    foreach reasonable slot number per machine $S$ do
        foreach feasible setting $O$ of $j$'s operator parameters do
            if $\mathfrak{C}(\langle j, O, \mathcal{P}, (S) \rangle) < C_{best}$ then
                $C_{\text{best}}, S_{\text{best}}, O_{\text{best}} = \mathfrak{C}(\langle j, O, \mathcal{P}, (S) \rangle), S, O$;
    $\mathcal{O} \leftarrow \mathcal{O} \cup \{(j, O_{\text{best}})\}$; $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{(j, S_{\text{best}})\}$;
return $\langle \mathcal{L}, \mathcal{O}, \mathcal{P}, \mathcal{Q} \rangle$;
```

```
opt_{O,P,Q}(L, T_max): L: physical plan template;
                       T_max: maximum completion time.
returns: lowest-cost deployment plan for L that runs under T_max.
$C_{\max}, D_{\text{best}} \leftarrow \infty, \perp$;
foreach machine type $m$ do
    $N \leftarrow \min\{N \mid \mathfrak{C}(\texttt{opt}_{\mathcal{O},\mathcal{Q}}(\mathcal{L}, (m, N)) \leq T_{\max}\}$ by exponential search;
    repeat
        $D, \check{c} \leftarrow \texttt{opt}_{\mathcal{O},\mathcal{Q}}(\mathcal{L}, (m, N))$;
        if $\mathfrak{C}(D) < C_{max}$ then $C_{\max}, D_{\text{best}} \leftarrow \mathfrak{C}(D), D$;
        ;
    until $\check{c} \geq C_{max}$;
return $D_{best}$;
```

Very briefly, the algorithm for searching for the optimal plan with x switches works by partitioning jobs in $\mathcal{L}$ into $x+1$ groups, where cluster switching takes place on partition boundaries. Once a plan is chosen for a partition, its time and cost update the time/cost budgets available to remaining partitions, which restrict the search for their plans. Details are omitted because of space constraints.

5 Other Issues

Iterative Programs For simplicity of presentation, we have not yet discussed loops. Cumulon supports a loop construct in the workflow of jobs. It is intended for iterative programs (not for writing low-level, C-style loops that implements matrix multiply, for example). As mentioned in Section 4.1, cost estimation for iterative programs is very difficult, because convergence depends on factors ranging from input characteristics (e.g., condition numbers of matrices) to starting conditions (e.g., warm vs. cold).

Cumulon takes a pragmatic approach in this regard. We focus on predicting the per-iteration time and cost of an iterative program. For short loop bodies (such as PLSI), we first unroll it to enable more optimization opportunities. The user can either reason with this per-iteration "rate" produced by Cumulon, or specify the desired number of iterations to get the overall cost. In practice, this limitation is not severe, because users often start with trial runs until they understand the convergence properties on full datasets. An interesting direction of future work is to look for cases where we can do more, e.g., when factors affecting convergence are understood and can be efficiently computed without running the program.

Coping with Performance Variance In Section 4.1.2, we have shown how to deal with variance in task completion time, which can result from transient behaviors or self-interference among tasks, and affect our prediction of expected job completion time. But this step is only a beginning. Performance uncertainty can arise for many reasons. It can stem from our optimizer's incomplete or inaccurate knowledge of the input properties: e.g., whether and where non-zero entries are clustered in a sparse matrix. In a cloud setting, for a (virtual) machine of given type, a cloud provider may assign an AMD instead of an Intel CPU, which affects the performance of linear algebra libraries. Depending on where the provisioned machines are located in the provider's data center, the network characteristics may vary. Performance can also be impacted by other virtual machines running on the same physical machine, and by other workloads sharing the same physical network.

While Cumulon is far from sophisticated in tackling performance variance from such sources, it offers a practical guard against wrong decision with its runtime monitoring support. When a deployment plan is executing, Cumulon monitors its performance and alerts the user as soon as it detects potentially significant deviations from the optimizer's time and cost predictions, e.g., when the first iteration takes much longer than expected. Upon receiving an alert, the user may decide to cancel the deployment to avoid sinking more money. As future work, we plan to investigate the possibility of dynamic reoptimization using data collected during ongoing execution.

6 Experiments

Our experiments are conducted on Amazon EC2. We consider the EC2 machine types listed in Table 1, and use the published rates (but without rounding up usage hours). The HDFS block size is 32MB; Cumulon's tiles have the same size by default (equivalent of a 2048 × 2048 dense double matrix). HDFS replication factor is set to 3 or the cluster size (N), whichever is smaller. These configuration settings do impact performance and reliability, but unlike the number of slots per machine (S), their choices are far less dependent on particular Cumulon programs. We have found these settings to work well for all workloads we tested, and therefore we do not include them in $\mathcal{Q}$ for plan optimization.

We use a variety of programs to evaluate Cumulon. 1) ***Matrix multiplies*** with varying dimensions and sparsity allow us to experiment with different input characteristics. 2) ***RSVD-1***, introduced in Section 1, is the dominating step in a randomized SVD algorithm. 3) ***GNMF-1***, also used by SystemML [8] for evaluation, is an expensive step in one iteration of the popular Gaussian Non-Negative Matrix Factorization algorithm. 4) ***PLSI-1*** is Line 3 in Algorithm 1, an expensive step in a PLSI iteration. 5) ***PLSI-full*** is a Hadoop program consisting of a non-Cumulon part, which downloads and preprocesses documents into the word-document matrix $\mathbf{O}$, followed by a Cumulon part, which performs PLSI on $\mathbf{O}$.

6.1 Execution Engine and Physical Operators

We now demonstrate the advantages of the executions strategies enabled by Cumulon's flexible model and physical operators as well as their efficient implementation on top of Hadoop/HDFS. Here, we focus on showing how, given a cluster, our approach achieves faster running time (and hence lower cost).

Comparison with SystemML First, we compare Cumulon with SystemML using a number of matrix multiply programs with varying sizes, shape, and sparsity. Their running times on the same cluster, broken down by jobs and phases, are shown in Figure 9. For SystemML, the number of map slots and that of reduce slots are always 2, and we choose the faster one of their RMM and CPMM strategies for each program. For Cumulon, we use the deployment plans picked by Cumulon; the number of (map) slots is also always 2. Table 2 shows the input dimensions and the split factors ended up being used by SystemML and Cumulon.

For Cumulon, we omit streaming granularity settings (Section 3.2) here and in subsequent discussion; in most cases choosing this granularity to be a tile works best, as CPU under-utilization can be compensated with a large enough S.

From Figure 9, we see that Cumulon is significantly faster than

	$\mathbf{A}$ $(m \times l)$	$\mathbf{B}$ $(l \times n)$	SystemML	Cumulon
	if sparse, sparsity shown in parentheses		split factors f_m, f_l, f_n	
1	24k × 24k	24k × 24k	$8, 1, 8$	$4, 4, 4$
2	1 × 100k	100k × 100k	$1, 1, 50$	$1, 10, 10$
3	100 × 100k	100k × 100k	$1, 1, 50$	$1, 10, 10$
4	160k × 100	100 × 160k	$8, 1, 8$	$8, 1, 8$
5	100 × 20000k	20000k × 100	$1, 80, 1$	$1, 80, 1$
6	200k × 200k (0.1)	200k × 1000	$100, 1, 1$	$10, 10, 1$
7	1000k × 1000k (0.01)	1000k × 1	$100, 1, 1$	$10, 10, 1$

Table 2: Choices of split factors by SystemML and Cumulon for $\mathbf{A} \times \mathbf{B}$.

SystemML (and hence proportionally cheaper in terms of monetary cost), for all but one program (#4). For #4, the two inputs are small and shaped like column and row vectors, where SystemML's RMM strategy happens to use the same (optimal) split factors as Cumulon; thus the two have comparable performance.

Two factors attribute to Cumulon's advantage: 1) Cumulon's map tasks do more useful work than SystemML's; 2) SystemML's choices of split factors are severely limited. We examine these factors in isolation later when discussing Figures 10 and 12.

More Complex Workflows and Advantage of *MaskMul* Figure 11 compares SystemML and Cumulon on two workflows, GNMF-1 and PLSI-1, on the same cluster. For PLSI-1, $m = 3375104$, $n = 131151$, $k = 100$. GNMF-1 computes $\mathbf{H} \circ (\mathbf{H}^\intercal \times \mathbf{V}) \circ/ (\mathbf{W}^\intercal \times \mathbf{W} \times \mathbf{H})$, where $\mathbf{H}$ is $k \times m$, $\mathbf{V}$ is $n \times m$, and $\mathbf{W}$ is $n \times k$ as in PLSI-1. There are two map slots per machine, and for SystemML, also two reduce slots. SystemML uses 5 full MapReduce jobs for GNMF-1, and 1 for PLSI-1; in contrast, Cumulon uses 4 and 1 *map-only* jobs, respectively. From Figure 11, we see that advantages of Cumulon carry over to more complex workflows.

Figure 11 further compares two plans for PLSI-1 in Cumulon: one that does not use the *MaskMul* operator (recall Section 3.2) and one that uses it. We see that performance improvement enabled by *MaskMul* is dramatic (more than an order of magnitude), further widening the lead over SystemML.[3]

Comparison with Other Hadoop-Based Implementations Going beyond SystemML, we examine more broadly how to implement matrix multiply with general split factors on top of Hadoop. We compare three options. 1) Cumulon: Two map-only jobs, where the first multiplies and the second adds. 2) ***Pure MapReduce***: Two map-reduce jobs, where the map tasks in the first job simply replicate and shuffle input to reduce tasks to be multiplied, like in SystemML. Unlike SystemML, however, flexible choices of split factors are allowed, so the second job may be required for adding. (This option enables us to study the impact of different uses of map tasks in isolation from the effect of different split factors.) 3) ***Cumulon+reduce***: One full map-reduce job, where the map tasks read directly from HDFS to multiply, like Cumulon; however, the results are shuffled to reduce tasks to be added.

Figure 10 compares the running times of these options in the same cluster, as well as settings of split factors and number of slots (optimized for respective options). First, we see that pure MapReduce incurs a significant amount of pure overhead in its map tasks; in contrast, other options' map tasks perform all multiplications. Second, the map-only approach of Cumulon allows it to focus all resources on map tasks, without the overhead and interference caused by having to shuffle to reduce tasks. For example, because of resource contention, Cumulon+reduce is forced to allocate 3 map slots for multiplication (vs. 4 for Cumulon) and use bigger split factors (which lead to more I/O). Also, general sorting-based shuffling supported by Hadoop is an overkill for routing data to matrix operators, who know exactly which tiles they need. Thus, in spite of the fact that writes to HDFS are slower than to local disks (for shuffling), Cumulon's approach of passing data through HDFS between map-only jobs is the better option.

6.2 Deployment Plan Space and Optimization

We begin with experiments that reveal the complexity of the search space and motivate the need for jointly optimizing settings of operator parameters ($\mathcal{O}$), hardware provisioning ($\mathcal{P}$), and configuration ($\mathcal{Q}$). These experiments also exercise the Cumulon optimizer. Then, we see examples of how Cumulon helps users make deployment decisions and works with a program with non-matrix Hadoop components. We also explore plans with cluster switching.

Impact of Operator Parameter and Configuration Settings We show that even if we are given a fixed cluster, different $\mathcal{O}$ and $\mathcal{Q}$ settings can have a big impact on performance. Figure 12 compares a number of deployment plans for a dense matrix multiply on the same cluster, while the number of slots per machine (S) varies. For each S setting, we consider the plan with the optimal $\mathcal{O}$ (split factors) for this S, as well as a plan with same split factors as SystemML's RMM (which requires $f_l = 1$). However, this latter plan is implemented in Cumulon and does not carry the overhead of SystemML's pure MapReduce implementation. (Thus, this plan helps reveal the impact of SystemML's suboptimal split factor choices in isolation from the overhead of its restricted use of map tasks.)

From Figure 12 we make two observations. First, poor choices of $\mathcal{O}$ and $\mathcal{P}$ lead to poor performance; e.g., the second plan for $S = 1$ is nearly three times more expensive than the optimal plan with $S = 6$. Second, the choices of $\mathcal{O}$ and $\mathcal{P}$ are interrelated. As S increases, memory available to each slot decreases, which obviously limits the input split size. However, the effect of S on the optimal split factors is more intricate than simply scaling them proportionally to make splits fit in memory, as we can see from Figure 12.

Also note that the optimal S setting in general is not the number of cores per machine (8 in this case). One of the reasons is that JBLAS, which we use for submatrix multiply, is multi-threaded, so even a smaller S can fully utilize the CPU; overcommitting hurts performance. In this case, automatic cost-based optimization works much more reliably than general rules of thumb.

Impact of Machine Type Next, we see how machine types influence optimality of deployment plans. Figure 13 explores various plans for dense matrix multiply on a 20-machine cluster of four types. There are four "base" plans, each optimized for one type. As the legend shows, they have rather different $\mathcal{O}$ and $\mathcal{Q}$ settings.

Figure 13 also shows what happens if one simply "retools" a plan optimized for one machine type for another. In particular, if the target machine type does not have enough memory run this plan, we lower the plan's S setting so that it can run without thrashing. This retooling is not always feasible; e.g., Plan 4, optimized for m1.xlarge, cannot be retooled for m1.small and c1.medium. In general, retooled plans are suboptimal, and in some cases (e.g., Plan 1 retooled for c1.xlarge) can incur twice as much cost as the optimal.[4]

Impact of Cluster Size Even if we fix the machine type, optimal settings for $\mathcal{O}$ and $\mathcal{Q}$ will change as we vary the cluster size N. In Figure 14, we consider RSVD-1 on increasing large clusters of c1.xlarge machines. First, we find Plan 1 as the best for $N = 1$, and Plan 2 as the best for $N = 250$. Their $\mathcal{O}$ and $\mathcal{Q}$ settings are shown in the legend. We retool the plans for different cluster sizes in a straightforward way: they simply run with their $\mathcal{O}$ and $\mathcal{Q}$ settings, so

[3] In more detail, *MaskMul* is applied to the second multiply (between $\mathbf{C} \times \mathbf{E}$ and $\mathbf{B}^\intercal$) under $\circ/$. This second multiply is where the bulk of the computation is—recall that $\mathbf{C}$ is $m \times k$, $\mathbf{E}$ is $k \times k$, $\mathbf{B}^\intercal$ is $k \times n$, and $m, n \gg k$.

[4] Figure 13 shows monetary cost, not time. As a side note on how to interpret such figures, having the lowest cost (Plan 2 and c1.medium in this case) may not imply the best deployment plan in the presence of tighter completion time constraints. Here, Plans 3 and 4 actually complete quicker, but are costlier because the rates of their machine types are higher.

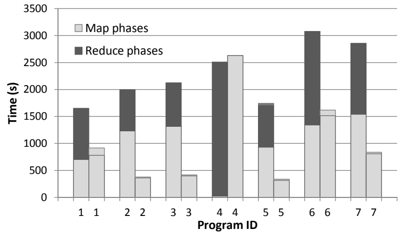

Figure 9: SystemML vs. Cumulon for matrix multiply programs (see table in text), on 10 m1.large machines. For each program, the left bar shows SystemML and the right bar shows Cumulon.

Time || slots ↘	Cumulon	pure MapReduce	Cumulon +reduce
1st map	1980s || 4	1885s || 2	2465s || 3
1st reduce	n/a	2759s || 4	924s || 2
2nd map	510s || 4	d/c†	n/a
2nd reduce	n/a	d/c†	n/a
split factors	$6, 6, 6$	$6, 6, 6$	$8, 8, 8$

Figure 10: Options for multiplying two dense 48k × 48k matrices using flexible split factors, on 10 c1.xlarge machines. († Implementation-dependent; irrelevant here as pure MapReduce's first job alone is already more costly than other options.)

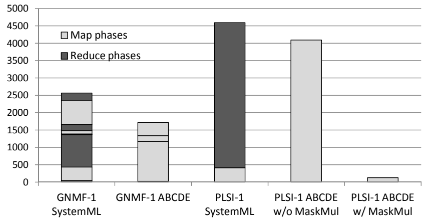

Figure 11: SystemML vs. Cumulon for GNMF-1 and PLSI-1, on 10 m1.large machines (some jobs/phases are short and thus may be difficult to discern from the figure).

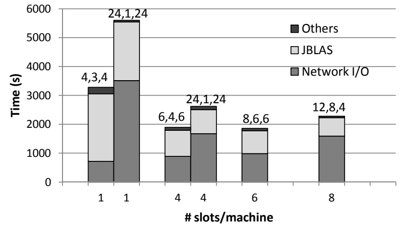

Figure 12: Effect of operator parameter and configuration settings (split factors and number of slots per machine) on running time of multiplying two 48k × 48k dense matrices. The cluster has 10 c1.xlarge machines. The split factors for *Mul* are shown on top of the respective bars.

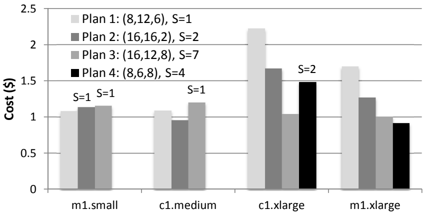

Figure 13: Effect of machine type on the cost of multiplying two 96k × 96k dense matrices. $N = 20$. The legend shows, for each base plan, its settings of the three split factors and the number (S) of slots per machine. Adjustment to S (if needed) is shown on top of the respective bar.

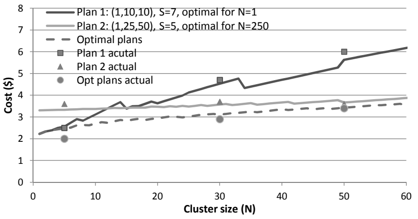

Figure 14: Effect of cluster size on the cost of RSVD-1 (with $l = 2\text{k}, m = 200\text{k}, n = 200\text{k}, k = 5$) on c1.xlarge machines. Cost curves are predicted; costs of actual runs for sample data points are shown for comparison.

work done by each task remains the same, and varying N translates to varying number of waves. The predicted costs of running these plans retooled from Plans 1 and 2 are plotted in Figure 14. For comparison, the figure also plots the predicted cost of the optimal plan found by our optimizer for every N.

As we can see from the figure, sticking to the same $\mathcal{O}$ and $\mathcal{Q}$ settings when N changes is suboptimal. Intuitively, Plan 1 performs more work per task and requires overall fewer tasks, but when N grows, many slots may idle in the last wave, leading to waste. On the other hand, Plan 2 has a lot of tasks, which are an overkill and carry more overhead and I/O when N is small. The optimal settings of $\mathcal{O}$ and $\mathcal{Q}$ change with N, and result in lower costs than Plans 1 and 2 (except for the two N's they are optimized for).

Since the cost curves in this figure are predicted by Cumulon, we also conduct a number of actual runs and plot their costs for comparison. These costs are consistent with our predictions.

Putting It Together We now show how Cumulon can help users make intelligent deployment decisions by generating Figure 1 seen earlier in Section 1. The figure visually presents the tradeoff between completion time and monetary cost[5] across different machine types, and enables users to ask "what-if" questions with different deadline and budget scenarios. Each data point of a cost curve is obtained by invoking the Cumulon optimizer to find the lowest-cost deployment plan for RSVD-1 that completes within the given time for the given machine type. This plan specifies not only parameter settings for its physical operators but also the cluster size and the number of slots per cluster. On a desktop machine with moderate hardware, it took under a minute to generate the whole figure (excluding the costs of actual runs for sample data points). This optimization overhead is well worth the benefit provided by Figure 1—one actual run can easily take hours to complete.

To see that Cumulon's cost estimation is effective, we also conducted actual runs for some data points; the results are largely in line with the predictions. Additional evaluation of cost estimation can be found in Figure 14 and figures in Section 4.1.

Cluster Switching and PLSI-Full Finding a simple program that warrants cluster switching turned out to be difficult. Cluster switching did not bring significant benefit to matrix multiplies, RSVD-1, GNMF-1, or PLSI-1, or complete versions of the last three algorithms. For example, in complete RSVD, while cluster switching at the end of RSVD-1 may seem obvious, it actually makes little difference because so little work remains after this dominating RSVD-1 step. Hence, our experience confirms our conjecture in Section 4.2 and justifies our approach of prioritizing the search for plans with few or no switches.

Nonetheless, to test our optimizer's support for cluster switching, we constructed a synthetic program with two dense matrix multiplies: the first one is between two 60k × 60k matrices; the second one is between 400k × 100 and 100 × 400k. Even for this program, one c1.medium cluster turns out to work well. Things become more interesting if we assume that c1.medium is no longer available. Among the remaining four machine types, c1.xlarge is most suited for the first multiply, while m1.small is best for the second. Our optimizer is able to find that the optimal plan involves a switch from c1.xlarge to m1.small when $T_{\text{max}} \geq 1.4$ hours. Under tighter completion time constraints, however, the cost of switching itself makes it unattractive. We omit the details.

Finally, we consider PLSI-full, which includes both non-matrix and matrix parts. The second part runs Algorithm 1 (3 iterations) and is handled by Cumulon. The first part consists of hand-optimized Hadoop jobs that download a document collection as separate `bz2` files (about 13GB total), decompress them, and extract a sample of text documents (3.5GB) form the collection to prepare a 9460k × 1095k (with sparsity of 2×10^{-5}) word-document matrix for the second part. Since Cumulon is built on top of Hadoop, it is easy to execute these two parts as a series of Hadoop jobs. While Cumulon optimizes the second part, we have to manually tune the first.

It turns out that while the first part of PLSI-full prefers a large number of small machines, the second part prefers fewer but larger

[5] Cost of data ingress/ogress depends on the setup, and is not shown here. Assuming Amazon EBS for stable storage, this cost would be no more than $2 (same for all deployment plans in Figure 1).

machines. Thus, we select two candidate clusters: C_1, with 20 c1.medium machines, and C_2, with 6 m1.xlarge machines. We consider three plans: 1) run the entire PLSI-full on C_1; 2) run the entire PLSI-full on C_2; 3) run the first part on C_1, switch to C_2, and run the second part. The results are summarized below.

	Time (s)				Cost ($)			
Plan	Preprocess	Switch	PLSI	Total	Preprocess	Switch	PLSI	Total
C_1	10757.5	0	13830	24587.5	9.90	0	12.72	22.62
C_2	14480.5	0	11712	26192.5	12.51	0	10.12	22.63
$C_1 \to C_2$	10757.5	670	11712	23139.5	9.90	0.58	10.12	20.59

Here, cluster switching reduces the cost from $22.62 to $20.59.

7 Related Work

A number of recent projects supporting statistical computing on big data subject scientists and statisticians to new, and in most cases, lower-level, programming abstractions. RHIPE [9], Ricardo [6], and Analytics' R/Hadoop enable R code to invoke Hadoop and vice versa, but programmers are responsible for parallelization and tuning. Apache Mahout and MADlib [5, 10] offer libraries of statistical analysis routines (for Hadoop and a parallel database system, respectively). Historically, library-based approaches have been either limited by hard-coded assumptions tied to particular data representations and execution environments, or ridden with knobs that are difficult to tune for new deployment settings.

Automatic, cost-based optimization is a staple of database systems. Indeed, MADlib leverages a database optimizer to alleviate tuning burden, but database optimizers are often inadequate for matrix-based workloads. Database-style optimization has also been applied to general MapReduce systems, e.g., by Starfish [11]. However, such work faces difficult challenges in modeling arbitrary map and reduce functions. Because Cumulon focuses on matrix-based programs, we are able to take a white-box approach specialized for such workloads. SystemML [8] takes an approach similar to ours, but as we have seen earlier, Cumulon avoids some significant overheads that SystemML incurs with its MapReduce-based implementation. Furthermore, SystemML only considers optimization of execution, while Cumulon also jointly considers hardware provisioning and configuration settings.

Automatic resource provisioning has been studied recently for general MapReduce systems [14, 23, 25, 12]. In contrast to the black-box workloads that these systems face, we focus on declaratively specified statistical workloads, which lead us to a different challenges and opportunities as we have shown in this paper.

On a high level, we share the goal of making big-data analytics easier with many recent systems. For example, projects such as SciDB [2] are rebuilding entire systems from the ground up for specific workload types. Targeting graph mining, PEGASUS [15] implements *generalized iterated matrix-vector multiply* efficiently on Hadoop. RIOT [24] and RevoScaleR focus on making statistical computing workloads in R I/O-efficient; pR [16] automatically parallelizes function calls and loops in R. Pig [17], Hive [20], and SciHadoop [4] are examples of higher-level languages and execution plan generators for MapReduce systems. Our work goes beyond these systems by addressing important usability issues of automatic hardware provisioning and configuration.

8 Conclusion and Future Work

We have presented Cumulon, a system that simplifies both the development and deployment of matrix-based data analysis programs in the cloud. We have shown how pure MapReduce is a poor fit for such workloads, and demonstrated how Cumulon's flexible execution model, new operators, and unconventional implementation on top of Hadoop/HDFS together provide significant performance improvements over existing Hadoop-based systems for statistical data analysis. We have motivated the need for automatic and joint optimization of not only physical operators and their parameter settings, but also hardware provisioning and configuration settings in the cloud. We have shown how a suite of benchmarking, simulation, modeling, and search techniques provide effective cost-based optimization over this vast space of possibilities.

We have already pointed out a number of directions for future work throughout this paper, such as more sophisticated handling of performance variance, exposing variance in cost estimation to users, and dynamic reoptimization. We also plan to consider the HPC offerings by Amazon EC2 and intelligently choose between HPC- and Hadoop/HDFS-based implementations. While we currently focus on addressing an individual user's need, it will be nice to extend Cumulon to optimize, schedule, and provision for workloads of multiple users programs in a shared cluster setting. Going a step further, it would be interesting to see, from a cloud provider's perspective, how a better understanding of matrix-based workloads can help in deciding what machine types to offer to support such workloads, and how to price these machine types sensibly.

References

[1] V. R. Borkar, Y. Bu, M. J. Carey, J. Rosen, N. Polyzotis, T. Condie, M. Weimer, and R. Ramakrishnan. Declarative systems for large-scale machine learning. *IEEE Data Engineering Bulletin*, 35(2):24–32, 2012.

[2] P. G. Brown. Overview of SciDB: Large scale array storage, processing and analysis. *SIGMOD 2010*.

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

[4] J. B. Buck, N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. A. Brandt. SciHadoop: Array-based query processing in Hadoop. *Supercomputing 2011*.

[5] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. MAD skills: New analysis practices for big data. *VLDB 2009*.

[6] S. Das, Y. Sismanis, K. S. Beyer, R. Gemulla2, P. J. Haas, and J. McPherson. Ricardo: Integrating R and Hadoop. *SIGMOD 2010*.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *OSDI 2004*.

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

[9] S. Guha. *Computing Environment for the Statistical Analysis of Large and Complex Data*. PhD thesis, Purdue University, 2010.

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

[11] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of MapReduce programs. *PVLDB Endowment*, 4(11):1111–1122, 2011.

[12] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics. *SoCC 2011*.

[13] T. Hofmann. Probabilistic latent semantic indexing. *SIGIR 1999*.

[14] K. Kambatla, A. Pathak, and H. Pucha. Towards optimizing hadoop provisioning for the cloud. *HotCloud 2009*.

[15] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A peta-scale graph mining system. *ICDM 2009*.

[16] J. Li, X. Ma, S. B. Yoginath, G. Kora, and N. F. Samatova. Transparent runtime parallelization of the R scripting language. *J. Parallel & Distributed Computing*, 71(2):157–168, 2011.

[17] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. *SIGMOD 2008*.

[18] V. Rokhlin, A. Szlam, and M. Tygert. A randomized algorithm for principal component analysis. *SIAM J. Matrix Analysis & Applications*, 31(3), 2009.

[19] S. Seo, E. J. Yoon, J.-H. Kim, S. Jin, J.-S. Kim, and S. Maeng. HAMA: An efficient matrix computation with the MapReduce framework. *CloudCom 2010*.

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

[21] S. Toledo. A survey of out-of-core algorithms in numerical linear algebra. In *DIMACS Series In Discrete Mathematics And Theoretical Computer Science: External Memory Algorithms*, pages 161–179. 1999.

[22] L. G. Valiant. A bridging model for parallel computation. *CACM*, 33(8), 1990.

[23] A. Verma, L. Cherkasova, and R. H. Campbell. ARIA: automatic resource inference and allocation for mapreduce environments. *ICAC 2011*.

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

[25] Z. Zhang, L. Cherkasova, A. Verma, and B. T. Loo. Meeting service level objectives of Pig programs. In *2012 Intl. Workshop on Cloud Computing Platforms*.