Cumulon: Cloud-Based Statistical Analysis from Users' Perspective

Botong Huang
Department of Computer Science
Duke University
bhuang@cs.duke.edu

Nicholas W.D. Jarrett
Department of Statistical Science
Duke University
nwj2@stat.duke.edu

Shivnath Babu
Department of Computer Science
Duke University
shivnath@cs.duke.edu

Sayan Mukherjee
Department of Statistical Science
Duke University
sayan@stat.duke.edu

Jun Yang
Department of Computer Science
Duke University
junyang@cs.duke.edu

Abstract

Cumulon *is a system aimed at simplifying the development and deployment of statistical analysis of big data on public clouds.* Cumulon *allows users to program in their familiar language of matrices and linear algebra, without worrying about how to map data and computation to specific hardware and software platforms. Given user-specified requirements in terms of time, money, and risk tolerance,* Cumulon *finds the optimal implementation alternatives, execution parameters, as well as hardware provisioning and configuration settings—such as what type of machines and how many of them to acquire.* Cumulon *also supports clouds with auction-based markets: it effectively utilizes computing resources whose availability varies according to market conditions, and suggests best bidding strategies for such resources. This paper presents an overview of* Cumulon *and the challenges encountered in building this system.*

1 Introduction

The ubiquity of the *cloud* today presents an exciting opportunity to make big-data analytics more accessible than ever before. Users who want to perform statistical analysis on big data are no longer limited to those at big Internet companies or HPC (High-Performance Computing) centers. Instead, they range from small business owners, to social and life scientists, and to legal and journalism professionals, many of whom have limited computing expertise but are now beginning to see the potential of the commoditization of computing resources. With publicly available clouds such as Amazon EC2, Microsoft Azure, and Google Cloud, users can rent a great variety of computing resources instantaneously, and pay only for their use, without worrying about acquiring, hosting, and maintaining physical hardware.

Unfortunately, many users still find it frustratingly difficult to use the cloud for any non-trivial statistical analysis of big data. ***Developing*** efficient statistical analysis programs requires tremendous expertise and effort. Most statisticians would much prefer programming in languages familiar to them, such as R and MATLAB,

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

where they can think and code naturally in terms of matrices and linear algebra. However, scaling programs written in these languages to bigger data is hard. The straightforward solution of getting a bigger machine with more memory and CPU quickly becomes prohibitively expensive as data volume continues to grow. Running on an HPC cluster requires hardware resources and parallel programming expertise that few have access to. Although the cloud has made it considerably easier than before to tap into the power of parallel processing using commodity hardware, popular cloud programming platforms, such as *Hadoop*, still require users to think and code in low-level, platform-specific ways. Emerging libraries for statistical analysis and machine learning on these platforms help alleviate the difficulty somewhat, but in many cases there exists no library tuned for the specific problem, platform, and budget at hand, so users must resort to extensive retooling and manual tuning. To keep up with constant innovations in data analysis, systems that support rapid development of brand new computational procedures are sorely needed.

Deploying statistical analysis programs in the cloud is also difficult. Users face a maddening array of choices, including hardware provisioning (e.g., *"should I get five powerful machines of this type, or a dozen of these cheaper, less powerful ones?"*), configuration (e.g., *"how do I set the numbers of map and reduce slots in Hadoop?"*), and execution parameters (e.g., *"what should be the size of each parallel task?"*). Furthermore, these deployment decisions may be intertwined with development—if a program is written in a low-level, non-"declarative" manner, its implementation alternatives and deployment decisions will affect each other. Finally, the difficulty of decision making is compounded by the emergence of auction-based markets, such as Amazon *spot instances*, where users can bid for computing resources whose availability is subject to market conditions. Current systems offer little help to users in making such decisions.

Many users have long been working with data successfully in their domains of expertise—be they statisticians in biomedical and policy research groups, or data analysts in media and marketing companies—but now they find it difficult to extend their success to bigger data because of lack of tools that can spare them from low-level development and deployment details. Moreover, while the cloud has greatly widened assess to computing resources, it also makes the risk of mistakes harder to overlook—wrong development and deployment decisions now have a clear price tag (as opposed to merely wasted cycles on some computers). Because of these issues, many potential users have been reluctant to move their data analysis to the cloud.

Cumulon is an ongoing project at Duke University aimed at simplifying the development and deployment of statistical analysis in the cloud. When developing such programs, we want users to think and code in a high-level language, without being concerned about how to map data and computation onto specific hardware and software platforms. Currently, *Cumulon* targets matrix computation workloads. Many statistical data analysis methods (or computationally expensive components thereof) can be expressed naturally with matrices and linear algebra. For example, consider *singular value decomposition* (*SVD*) of matrices, which has extensive applications in statistical analysis. In the randomized algorithm of [18] for computing an approximate SVD, the first (and most expensive) step, which we shall refer to as RSVD-1, 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 $\mathbf{G} \times (\mathbf{A} \times \mathbf{A}^\intercal)^k \times \mathbf{A}$.

When deploying such programs, users should be able to specify their objectives and constraints in straightforward terms—time, money, and risk tolerance. Then, *Cumulon* will present users with the best "plans" meeting these requirements. A plan encodes choices of not only implementation alternatives and execution parameters, but also cluster resource and configuration parameters. For example, Figure 1 shows the costs[1] of best plans for RSVD-1 as we vary the constraint on expected completion time. In general, the best plans differ across points in this figure; choosing them by hand would have been tedious and difficult. The figure reveals a clear trade-off between completion time and cost, as well as the relative cost-effectiveness of various machine types for the given program, making it easier for users to make informed decisions.

[1]To simplify the interpretation of results in this paper, we do not follow Amazon EC2's practice of rounding usage time to full hours when computing the costs reported here (though it is straightforward for *Cumulon* to use that pricing policy).

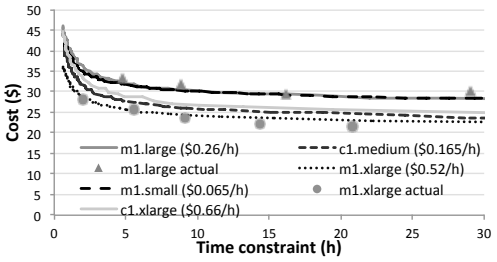

Figure 1: (From [10]) Costs of the optimal deployment plans for RSVD-1 (with $l = 2{,}000$, $m = n = 200{,}000$, $k = 5$) using different Amazon EC2 machine types under different time constraints. Curves are predicted; costs of actual runs for sample data points are shown for comparison.

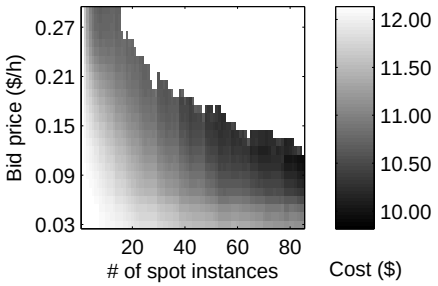

Figure 2: (From [11]) Estimated expected costs of the optimal plans for RSVD-1 (with $l = 2{,}000$, $m = n = 160{,}000$, $k = 5$), as we vary the bid price and number of spot instances to bid for. The cost is shown using intensity, with darker shades indicating lower costs. In the upper-right region, plans with the given bidding strategies fail to meet the user-specified risk tolerance; therefore, this region shows the baseline cost of the optimal plan with no bidding. All machines are of type `c1.medium`.

Cumulon is also helpful when working with an auction-based market of computing resources. For example, to encourage utilization of spare capacity, Amazon lets users bid for *spot instances*, whose *market prices* change dynamically but are often much lower than the regular fixed prices. Users pay the market price of a spot instance for each time unit when it is in use; however, as soon as the market price exceeds the bid price, the spot instance will be taken away.[2] For example, consider again RSVD-1.[3] It takes a baseline cost of $\$12.12$ to run under 28 hours, using 3 machines of type `c1.medium` at the fixed price of $\$0.145$ per hour. Taking advantage of additional spot instances, we can potentially complete faster and end up paying less overall than the baseline. Because of the uncertainty in future market prices, *Cumulon* lets users specify a risk tolerance—e.g., *"with probability no less than* 0.9*, the overall cost will not exceed the baseline by more than* 5%*."* Figure 2 shows the expected cost of optimal plans for RSVD-1 that meet the given risk tolerance, under various bidding strategies. Using information in this figure, *Cumulon* can recommend that user bid for additional 77 `c1.medium` spot instances at $\$0.11$ per hour each (versus the current market price of $\$0.02$), which would reduce the expected overall cost to $\$10.00$ while staying within the user's risk tolerance. Later in this paper, we will discuss how *Cumulon* makes such recommendations.

Challenges An analogy between *Cumulon* and a database system is useful. Much like a database system, *Cumulon* starts with a *logical plan* representing the input program, expressed in terms of well-understood matrix primitives—such as multiply, add, transpose, etc.—instead of relational algebra operators. Then, *Cumulon* applies cost-based optimization to find optimal *physical plans* (or simply *plans*, if the context is clear) implementing the given logical plan. Despite this high-level similarity, however, a number of unique challenges take *Cumulon* well beyond merely applying database techniques to cloud-based matrix computation.

To begin, there is a large and interesting design space for storage and execution engines in this setting. We have three pillars to build on: 1) the database approach, whose pipelined, operator-based execution makes it easy to discern semantics and process out-of-core data; 2) the data-parallel programming approach, represented by *MapReduce* [6], whose simplicity allows it to scale out to big data and large commodity clusters common to the cloud; and 3) the HPC approach, which offers highly optimized libraries for in-core matrix computation. Each

[2] Amazon EC2 actually does not charge for partial hour of usage of spot instances if they are taken away (as opposed to terminated voluntarily by users). This policy is specific to Amazon and can lead to some rather interesting bidding schemes. To avoid making our results too specific to Amazon, we always consider fractional hours when computing costs in this paper and make no special case for spot instances. (Again, *Cumulon* can readily support the actual Amazon policy instead.)

[3] Please note that there are minor differences between the workload settings and prices in the examples and figures of this paper, because the results were originally reported in different papers [10, 11] and obtained at times when Amazon charged different prices.

approach has its own strengths and weaknesses, and none meets all requirements of *Cumulon*. When developing *Cumulon*, we made a conscious decision to avoid "reinventing the wheel," and to leverage the popularity of existing cloud programming platforms such as Hadoop. Unfortunately, MapReduce has fundamental limitations when handling matrix computation. Therefore, *Cumulon* must incorporate elements of database- and HPC-style processing, but questions remain: How effectively can we put together these elements within the confines of existing platforms? Given the fast-evolving landscape of cloud computing, how can we reduce *Cumulon*'s dependency on specific platforms?

Additional challenges arise when supporting *transient nodes*, which refer to machines acquired through bidding, such as Amazon spot instances, with dynamic, market-based prices and availability. Getting lots of cheap, transient nodes can pay off by reducing the overall completion time, but losing them at inopportune times will lead to significant loss of work and higher overall cost. Although current cloud programming platforms handle node failures, most of them do not expect simultaneous departures of a large number (and easily the majority) of nodes as a common occurrence. Any efforts to mitigate the loss of work, such as copying intermediate results out of transient nodes, must be balanced with the associated overhead and potential bottlenecks, through a careful cost-benefit analysis. *Cumulon* needs storage and execution engines capable of handling massive node departures and supporting intelligent policies for preserving and recovering work on transient nodes.

Optimization is another area ripe with new challenges. Compared with traditional query optimization, *Cumulon*'s plan space has more dimensions—from settings of hardware provisioning and software configuration, to strategies for bidding for transient nodes and preserving their work. *Cumulon* formulates its optimization problem differently from database systems, as it targets user-centric metrics of time and money, as opposed to system-centric metrics such as throughput. Uncertainty also plays a more important role in *Cumulon*, because the cloud brings more uncertainty to cost estimation, and because *Cumulon* is user-facing: while good average-case performance may be acceptable from a system's perspective, cost variability must also be considered from a user's perspective. Therefore, *Cumulon* lets users specify their risk tolerance in optimization, allowing, for example, only plans that stay within budget with high certainty.

To support this uncertainty-aware, cost-based optimization, *Cumulon* faces many challenges in cost estimation that are distinct from database systems. While cardinality estimation is central to query optimization, it is less of an issue for *Cumulon*, because the sizes of matrices (including intermediate results) are usually known at optimization time. On the other hand, modeling of future market prices and modeling of placement of intermediate results become essential in predicting the costs of plans involving transient nodes. *Cumulon* needs to build uncertainty into its modeling, not only to support the risk tolerance constraint in optimization, but also to help understand the overall behavior of execution. For example, quantifying the variance among the speeds of individual threads of parallel execution improves the estimation of overall completion time.

In the rest of this paper, we give an overview of how *Cumulon* deals with these challenges, and then conclude with a discussion of related and future work. For further details on *Cumulon*, please see [10, 11].

2 Storage and Execution

Cumulon provides an abstraction for distributed storage of matrices. Matrices are stored and accessed by *tiles*, where each tile is a submatrix of fixed (but configurable) dimension. For large matrices, we choose a large tile size (e.g., 32MB) to amortize the overhead of storage and access, and to enable effective compression.

At a high level, *Cumulon*'s execution model has much in common with popular data-parallel programming platforms. A *Cumulon* program executes as a workflow of *jobs*. Each job reads and writes a number of matrices; input and output matrices must be disjoint. Dependent jobs execute in a 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, and write disjoint parts of output matrices. Data are passed between jobs only through the global, distributed storage.

Cumulon configures each available *node* (machine) into a number of *slots*, and schedules each slot to execute one task at a time. If a job consists of more tasks than slots, it will take multiple waves to finish. Elements of both database- and HPC-style processing are incorporated into task execution. *Cumulon* executes each task as a directed acyclic graph of physical operators in a pipelined fashion, much like the iterator-based execution in database systems. However, the unit of data passing between physical operators is much bigger than in database systems—we pass tiles instead of elements, to reduce overhead, and to be able to use the highly tuned BLAS library on submatrices. Physical operators can choose to batch multiple tiles in memory to create larger submatrices, which increase CPU utilization at the expense of memory consumption. For example, *Cumulon*'s matrix multiply operator provides a *streaming granularity* parameter that can be adjusted (automatically by *Cumulon*'s optimizer) to achieve the best trade-off for a given hardware configuration.

MapReduce No, Hadoop Yes It is worth pointing out that *Cumulon* does not follow the popular MapReduce model. In *Cumulon*, different tasks can read overlapping parts of the input data, directly from the distributed storage; there is no requirement of disjoint input partitioning or *shuffle*-based data passing as in MapReduce. This flexibility is crucial to efficient matrix computation, which often exhibits access patterns awkward to implement with MapReduce. In [10], we have demonstrated that *Cumulon*'s execution model enables far more efficient implementation choices of matrix workloads than approaches based on MapReduce.

For example, consider multiplying two big matrices $\mathbf{A}$ and $\mathbf{B}$. On a high level, virtually all efficient parallel matrix multiply algorithms work by dividing $\mathbf{A}$ and $\mathbf{B}$ into submatrices of appropriate dimensions: first, pairs of submatrices from the two inputs are multiplied in parallel; second, the results of the multiplies are grouped (by the output submatrix position they contribute to) and summed. In general, one input submatrix may be needed by multiple tasks in the first step. Under MapReduce, we would have to use a map phase to replicate the input submatrices and shuffle them to the reduce phase to be multiplied. Then, we need another round of map and reduce for the summation. In both steps, the map phase performs no useful computation. There are workarounds that address some aspects of this inefficiency by choosing submatrices consisting of entire rows and columns, but such submatrix dimensions are often suboptimal, leading to low compute-to-I/O ratios. In contrast, *Cumulon* supports fully flexible choices of submatrix dimensions, using simply one job for submatrix multiplies and one for summation, both performing useful work. *Cumulon* can further reduce overhead by folding the second job into a subsequent one that uses the result of $\mathbf{A} \times \mathbf{B}$, an optimization that we will come back to in Section 3.

Despite MapReduce's inadequacy, we still want to leverage the popularity of MapReduce-based platforms. Hadoop was an obvious choice when we started the *Cumulon* project, because it already had a healthy, growing ecosystem of developers, users, and service providers. Although *Cumulon*'s storage and execution models differ from MapReduce, we managed to implement *Cumulon* on top of Hadoop and HDFS without changing their internals. Specifically, every *Cumulon* job is implemented as a map-only job in Hadoop that can access specific tiles directly through a distributed tile storage implemented on HDFS. For example, Figure 3 illustrates how *Cumulon* implements $\mathbf{A} \times \mathbf{B} + \mathbf{C}$ on Hadoop and HDFS. Here, the summation step of $\mathbf{A} \times \mathbf{B}$ has been folded into the job that adds $\mathbf{C}$. Building on top of Hadoop and HDFS, *Cumulon* is able to inherit their features of scalability, elasticity, and failure handling; it is also easy to support complex workflows that combine *Cumulon*-powered matrix computation with regular Hadoop jobs for data wrangling and processing.

Finally, we note that *Cumulon*'s storage and execution models are simple and generic by design, so we can potentially implement new instances of *Cumulon* on top of other platforms such as *Spark* [26] and *Dryad* [12].

Support for Transient Nodes Making effective use of transient nodes requires attention to multiple points throughout *Cumulon*. For storing intermediate results, *Cumulon* does not assume that a distributed storage service (such as Amazon EBS or S3) exists independently of the provisioned nodes. Local storage attached directly to the nodes are cheaper and faster, but a sudden departure of transient nodes will lead to loss of valuable intermediate results. To ensure progress of execution, *Cumulon* provisions a set of *primary nodes*, which are

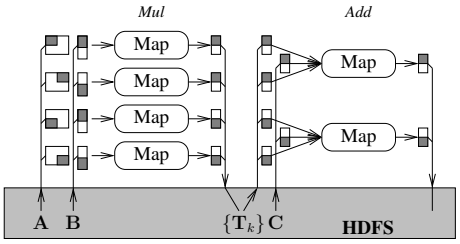

Figure 3: (From [10]) *Cumulon*'s map-only Hadoop jobs for $\mathbf{A} \times \mathbf{B} + \mathbf{C}$. Shaded blocks represent submatrices. $\{\mathbf{T}_k\}$ is the collection of results of from submatrix multiplies, yet to be grouped and summed.

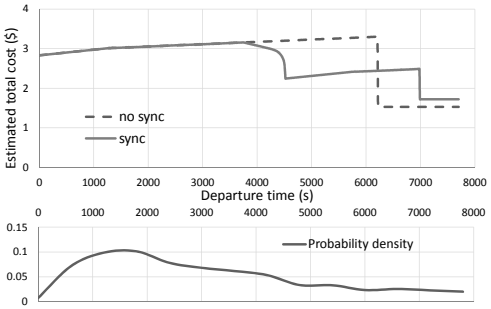

Figure 4: (From [11]) *Top*: Estimated costs of two plans (with and without *sync*) for a chain of five matrix multiplies. The *sync* job is applied to result of the third multiply. All matrices have dimension $30{,}000 \times 30{,}000$. There are 3 primary and 10 transient nodes, all of type `c1.medium`. *Bottom*: Probability density function for the distribution of the departure time of transient nodes, with bid price of $\$0.05$ and current market price of $\$0.02$. The model is based on the historical prices of Amazon spot instances.

regular machines acquired at fixed prices whose availability is not subject to market conditions. A naive strategy for preventing loss of work is for the transient nodes to write all intermediate results to primary node storage, but doing so is costly and can overwhelm the primary nodes (especially since we often use more transient nodes than primary ones). In general, it is difficult for the storage layer to decide intelligently what data to save, because that decision depends on when the data will be used, when the transient nodes are expected to depart, and how expensive it is to recompute the data. Therefore, *Cumulon* takes a cross-layer approach. In the storage layer, *Cumulon* caches data opportunistically—whenever a primary node accesses data from a transient node (or vice versa). Additionally, *Cumulon*'s execution engine supports a *sync* operator, which, for a given matrix, copies any tiles not already stored by the primary nodes from the transient nodes in a distributed fashion. *Cumulon*'s optimizer decides when and for which matrices *sync* is needed.

Support for transient nodes also necessitates support for recovery and heterogeneous clusters. Recovery is needed following the departure of the transient nodes. During recovery, *Cumulon* automatically identifies missing tiles required to complete the execution, and uses the lineage information inferred from the execution plan to recompute these tiles. Heterogeneity is needed because *Cumulon* may acquire transient nodes of a type different from that of the primary nodes (and may do so after the execution starts). It turns out that the optimal size of a task depends on what hardware it runs on. Therefore, *Cumulon* has an online task scheduler that dynamically assigns tasks of different sizes according to the available node types. The scheduler maintains a map for each output matrix, and, upon request to assign a task to a node, splits off a region of responsibility whose dimension best matches the optimal task size for the node type.

Again, we are able to implement support for transient nodes in *Cumulon* on top of Hadoop and HDFS. We use a custom Hadoop scheduler. The distribution tile storage runs on top of two HDFS instances—one involving all primary nodes and the other involving all transient nodes—and caches data between the instances upon access. The dual-HDFS approach is clean and resilient against the massive departure of all transient nodes; using a single HDFS would require non-trivial modification to its replication logic in order to provide a similar level of resiliency. The *sync* operator and recovery phase are both implemented as Hadoop jobs.

Figure 4 (focus on the top figure for now) illustrates the benefit of transient nodes and importance of *sync* for a simple workload consisting of a series of five matrix multiplies. The figure shows how the costs of two different plans (with and without *sync*) vary as the departure time of the transient nodes changes. Here, *sync* applies to the result of the third multiply. The two curves start from the same point on the left, where the cost reflects the baseline cost when no transient machines are used (here, they depart immediately and incur no extra cost). If both plans run to completion without transient nodes departing, their costs (shown on the right where

the curves become flat) would be much lower than the baseline, showing the benefit of transient nodes (in that case, the *sync* plan actually takes longer and costs slightly more than the no-*sync* plan, because of the overhead of *sync* itself). On the other hand, if the transient nodes depart during its execution, the no-*sync* plan would cost more than the baseline, as it loses valuable work done on the transient nodes. Fortunately, *sync* comes to the rescue. If the transient nodes depart between ≈ 4000s (when *sync* starts) and ≈ 6100s into the execution (when the no-*sync* plan would complete), the *sync* plan will cost less than the no-*sync* plan. As long as the departure occurs after ≈ 4200s (when *sync* is almost complete), the *sync* plan will cost less than the baseline. Of course, the departure probability of transient nodes is not uniform over time; we must account for this uncertainty when comparing these plans and the baseline. We will revisit this example when discussing optimization in Section 3.

3 Optimization

A simpler version of the optimization problem, used by our first iteration of *Cumulon* in [10] for plans that use no transient nodes, ignores risk tolerance. The resulting optimization is bi-criteria, involving time and money. One possible formulation is to find the plan with the lowest expected monetary cost among those expected to complete under a user-specified time constraint. The optimizer is given the logical plan representing the program, as well as input data characteristics (such as dimensions and sparsity of matrices). Thanks to the nature of matrix computation, performance uncertainty is manageable for many workloads, so this simple approach is practical.

Once we consider transient nodes, however, the degree of uncertainty becomes significant enough to affect user decisions. Therefore, we extend the optimization problem as follows. Let C denote be the expected monetary cost of the best plan (without using transient nodes) obtained by solving the simpler version of the optimization problem under the user-specified time constraint. *Cumulon* then looks for the plan (with possible use of transient nodes) having the lowest expected monetary cost, subject to the constraint that the actual cost of the plan exceeds δC with probability less than ϵ; here, (δ, ϵ) is the user-specified risk tolerance.

We clarify what we mean by a "plan with possible use of transient nodes" (although we omit the formal definition of the plan space here). First, because the optimization decision is based on the current market condition, the plan is intended to begin executing now, with a set of primary nodes. Optionally, the plan can bid for a number of transient nodes at a particular price, either immediately or at some point later during the execution. If the bid is successful, the plan will execute on both primary and transient nodes, with *sync* jobs at predetermined points in the execution. If the transient nodes depart before the execution completes, the primary nodes enter a recovery phase to bring the execution to a state consistent with one that would have been achieved by completing all jobs started by the partial execution on the primary nodes alone. Normal execution then resumes. The optimizer is responsible for choosing the primary node cluster, the transient node cluster (if any) and its bid price and time, as well as the execution plan (with *sync* jobs if applicable).

Note that in the above, we consider only a *single* round of decision about whether, when, and how to bid for transient nodes. The benefit of optimizing further ahead into the future (e.g., whether to bid for more transient nodes in additional rounds) is unclear, because market uncertainty widens quickly over a longer time horizon. A better approach, which we are actively investigating for *Cumulon*, is to optimize and adapt *online* as we continue to monitor execution progress and market conditions. With some extension, the single-round optimization problem above could serve as a useful building block for the online optimizer—at any point during execution, we can solve this problem for the remaining jobs to see whether we should wait or bid for transient nodes now. Another extension orthogonal to the use of transient nodes is cluster switching: we can partition a program into parts to be handled by different clusters best suited for their execution, with necessary data transfers between the clusters. *Cumulon* already considers and supports switching of the primary node cluster [10].

In the following, we briefly discuss cost estimation techniques and optimization algorithms in *Cumulon*.

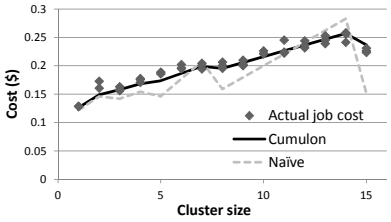

Figure 5: (From [10]) Actual job costs vs. predictions from the naive method and *Cumulon*. The job multiplies two dense matrices of sizes $12{,}000 \times 18{,}000$ and $18{,}000 \times 30{,}000$, and consists of 30 tasks. Machine type is `c1.medium`, with 2 slots per node.

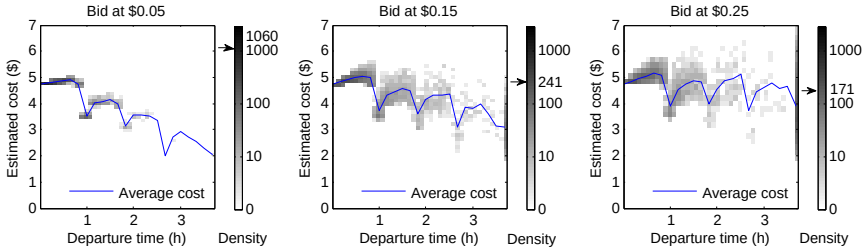

Figure 6: (From [11]) Effect of bidding price on the distribution of plan cost. Here, the workload consists of 12 jobs implementing two iterations of Gaussian non-negative matrix factorization. The baseline plan runs on a 3-node `c1.medium` cluster at the fixed price of $\$0.145$ per hour per node. We bid for a 10-node `c1.medium` cluster; the current market price is $\$0.02$ per hour per node. The density plots are generated from 5000 price traces. To make it easier to compare the three plots, we use the same (logarithmic) scale for all density maps, and use an arrow next to each map to mark the highest density in the corresponding plot.

Cost Estimation We begin by briefly describing how *Cumulon* estimates the completion time of a task. The two main contributors to task completion time are computation and network I/O. For each physical operator, we build a model to predict its computation time given machine type, number of slots per machine, operator parameter settings, input size, and input sparsity. *Cumulon* currently trains these models by benchmarking all machine types, though it would be helpful (and straightforward) to supplement training with performance traces of actual executions. We estimate the network I/O cost using an operator-specific model of the I/O volume, and a model for converting the I/O volume to time given the cluster sizes and configurations for primary and transient nodes. When modeling task completion time, we consider not only the expected time, but also variance—which is important in estimating job completion time, as explained below.

As explained in Section 2, *Cumulon* divides a job into multiple tasks and execute them in waves on all available slots. A naive estimate of job completion time would be the product of task completion time and the number of waves. However, this method works well only for a homogeneous cluster and when variance in task completion time is low. For example, Figure 5 shows the actual versus predicted costs of a matrix multiply job running on a homogeneous cluster (where cost equals the product of completion time, cluster size, and the per-node rate). Even in this simple setting, the naive estimates are inaccurate—the estimated cost fluctuates wildly as the cluster size varies, because the number of waves, an integral value, changes abruptly at specific cluster sizes. In contrast, the actual costs show a much smoother trend. The reason is that when task completion times vary, the scheduler assigns new tasks to slots that complete earlier, hence blurring the boundaries between waves and making the job completion times smoother. In predicting job completion times, *Cumulon* simulates the behavior of its scheduler for variable task completion times and potentially heterogeneous clusters. As Figure 5 exemplifies, the results are much more accurate than the naive estimates.

Optimization involving transient nodes requires costing *sync* jobs and the recovery phase. In both cases, *Cumulon* needs to estimate the number of tiles of a given matrix that are stored only at the transients nodes (and hence need to be copied or recovered). We derive this estimate by assuming that each tile is accessed by a node independently at a probability proportional to the processing power of the node, as more powerful nodes can complete more and/or bigger tasks and therefore access more tiles. The final estimate thus depends on the relative processing capacities of the primary and transient node clusters (a bigger transient cluster means fewer tiles will be stored at the primary nodes), as well as how many times an intermediate result is accessed in subsequent jobs (more accesses lead to more tiles cached at the primary nodes). To estimate recovery cost, *Cumulon* also needs to estimate the number of input matrix tiles required for recomputing a given subset of the output matrix tiles. To derive this estimate, we analyze input-output lineage for each operator, and assume that

the subset of tiles to be recovered is random.

Finally, *Cumulon* needs a model predicting the market price of transient nodes. Any stochastic model can be plugged in, provided that *Cumulon* can efficiently simulate the stochastic process. Currently, *Cumulon* uses a model with two components. First, we use non-parametric density estimation to approximate the conditional distribution of the future prices given the current and historical prices. To capture periodicity, we wrap the time dimension in a one-week cycle. Second, price spikes and their inter-arrival times are modeled as being conditionally independent given current and historical prices. We train the components using the historical prices of Amazon spot instances, and then combine the components to create a sample path for the stochastic price process. With this model, we can derive the distribution of the departure time of the transient nodes with a given bid price under the current market conditions. The bottom of Figure 4 plots an example departure time distribution. Using this distribution, we can then quantitatively compare the plans with and without *sync*, shown in the top of Figure 4, in terms of their expected costs and whether they meet the user-specified risk tolerance.

Optimization Algorithms As discussed in Section 1, *Cumulon* needs to explore a large, high-dimensional plan space. Although most *Cumulon* workloads are expensive enough to justify spending more time on optimization, we are still mindful of the optimization overhead, especially since we plan to make optimization online in the future. Thus, we use a number of techniques and assumptions to tame the complexity of optimization. First, *Cumulon* conducts a series of rule-based rewrites (e.g., linear algebra equivalences) on logical plans so that they lead to generally better physical plans with less computation and I/O or higher degree of parallelism. *Cumulon* then translates each logical plan into one or more alternative workflows of physical operators grouped into jobs; fewer jobs are usually preferred because of the overhead of initiating jobs during execution. For example, recall Figure 3: the merging of the second step of matrix multiply and the ensuing matrix add into a single job is carried out during this translation.

Given a job workflow, *Cumulon* still has to make many decisions to turn it into an executable plan. We solve the simpler optimization problem for plans without transient nodes by considering each possible machine type for the (primary node) cluster. Given the machine type, we bound the range of feasible cluster sizes and perform an exponential search for the optimal cluster size, exploiting the properties of the cost functions. For example, one such property is that as the cluster size increases, the execution time is generally non-increasing, while the cost is generally non-decreasing because of the growing parallelization overhead. As for the number of slots per machine, its choices are naturally limited on a given machine type, so we consider them exhaustively. Given the cluster configuration, we can then determine the optimal task size and other execution parameters for each physical operator independently. Finally, we consider cluster switching options. To bound the optimization time, we prioritize the search to explore plans with fewer switches first, and terminate the search when time runs out.

Starting with the baseline plan selected by the procedure above, *Cumulon* further optimizes the use of transient nodes. We consider each possible machine type for the transient nodes in turn. Beginning with the market price, we try successively higher bid prices in small increments ($\$0.01$ per hour). This simple search strategy is feasible because, fortunately, the bid price cannot go high under the risk tolerance constraint. Nonetheless, it is interesting to note that bidding *above* the regular, fixed price for the same machine type could be beneficial, because transient nodes are only charged at their market price, and a higher bid price will decrease the probability of their untimely departure. In general, it is difficult to make informed bids without *Cumulon*'s help. To see the effect of bid price more clearly, consider Figure 6, which compares the resulting cost distributions under three different bid prices. *Cumulon* computes these distributions using a collection of price traces obtained by repeatedly simulating from the stochastic market price model. With a low bid price, we can hope to get lucky and keep the transient nodes long enough to lower the cost; however, as shown by the high-density region on the left of Figure 6a, we are more likely to lose the transient nodes soon and end up paying more. With a high bid price, as shown in Figure 6c by the clear shift of density towards right, we are expected to have the transient nodes longer; however, we may not be able to improve the overall cost, because we also pay more for the tran-

sient nodes on average (as the market price can linger longer close to the higher bid price). The middleground (Figure 6b) between the two extremes may in fact be more attractive.

The decisions on the time and number of nodes to bid can be equally tricky for users. *Cumulon* considers bid times in small increments (10 minutes) until uncertainty becomes too high. *Cumulon* bounds the transient node cluster size by analyzing the parallelization overhead (as with the case of bounding the primary node cluster size), and by applying the risk tolerance constraint—even at low bid and market prices, a large number of transient nodes pose the risk of an untimely departure that may result in substantially higher cost than the baseline. Finally, *Cumulon* greedily adds *sync* jobs to the baseline plan: the *sync* job that gives the biggest improvement in expected cost will be picked, and the process repeats until no more improvement can be made.

In summary, *Cumulon* features a cost-based, uncertainty-aware optimizer that needs to deal with a large plan space. It takes a sampling-based approach towards handling uncertainty, and it keeps the optimization time under control using a combination of heuristics, practical constraints, and search techniques such as branch-and-bound. For example, on a regular laptop with an Intel i7-2600 CPU and 8G of memory, for RSVD-1 with 10 jobs, *Cumulon* is able to solve the full-fledged optimization problem—which involves finding optimal plans with bid times up to about one day into future, computing their expected costs, and quantifying the associated uncertainty in order to test whether they meet the given risk tolerance—under just one minute [11].

4 Related Work

Many projects are aimed at supporting statistical computing in the cloud. Some expose users to lower-level parallel programming abstractions, e.g., *RHIPE* [8], *Ricardo* [5], Revolution Analytics' *R/Hadoop*, and *MLI* [21]. Some provide higher-level libraries, e.g., Apache *Mahout* and *HAMA* [19]. Such libraries significantly simplifies development if they happen to offer the exact same algorithms needed by the task at hand, but they provide little help with the development of new algorithms beyond reusing implementations of low-level primitives. Furthermore, most of these libraries still have performance knobs that are difficult to tune for users. In contrast, by automatically parallelizing and optimizing programs expressed in terms of matrix primitives, *Cumulon* can simplify the development and deployment of even brand new algorithms.

Besides *Cumulon*, a number of other projects also support automatic optimization of statistical computing workloads. For example, *MADlib* [4, 9] leverages optimization techniques for parallel databases. *SystemML* [7, 2] takes an approach similar to ours, but only considers optimization of execution, while *Cumulon* additionally considers hardware provisioning and configuration setting as well as bidding strategies for transient nodes. The root of this difference lies in *Cumulon*'s focus on the users' perspective: while other systems optimize execution time on a given cluster, *Cumulon* goes a step further and asks what cluster would be most cost-effective (and how to bid for it) in the first place. Although automatic resource provisioning has been studied recently for general MapReduce systems [13, 23, 29], our focus on declaratively specified matrix-based programs leads to different challenges and opportunities in joint optimization of provisioning and execution that better exploits program semantics.

There has also been a lot of related work from the HPC community on automatic optimization of matrix-based programs, though none has gone as far as automatically making provisioning and configuration decisions. Most closely related to *Cumulon* is the work on "semantic" optimization of MATLAB programs (i.e., at the level of linear algebra operators) [17]. At one point, MATLAB provided a chain-matrix-multiply function that reordered multiplies using associativity. Generally speaking, however, platforms such as MATLAB and R do not automatically perform high-level linear algebra rewrites.[4] Beyond optimization at the level of linear algebra

[4]One reason why these platforms do not support linear algebra rewrite is the dynamic, interpreted nature of these languages. Another reason is the concern for reproducibility: for example, although in theory matrix multiplies are associative, in practice different orders can lead to different results due to errors inherent in the machine representation of numerical values. In *Cumulon*, we take the view that such concerns should not preclude automatic optimization by default—only in cases where such errors matter should users disable

expressions, there is also a long line of work from the compiler community on optimizing array code using loop analysis and transformations. Indeed, we have applied compiler techniques in a precursor project to *Cumulon* called *RIOT* [27, 28] to improve the I/O efficiency of matrix-based programs. Although *Cumulon* currently does not optimize at the loop level, how to exploit both styles of optimization in *Cumulon* remains an interesting possibility. We refer interested readers to [27, 28] for discussion of related work in this area.

Support for auction-based markets of cloud computing resources has also attracted much attention. A number of systems support using transient nodes for MapReduce: e.g., Amazon's own *Elastic MapReduce*, Qubole's auto-scaling clusters, and [15, 3]. They focus on storage and execution support; a more in-depth comparison with the Cumulon's dual-HDFS approach can be found in [11]. They do not help users choose bidding strategies or optimize their programs. Bidding strategies have been studied in [22, 24, 1, 25]; some of the approaches also models market price stochastically. However, unlike *Cumulon*, they assume black-box jobs, and hence do not have the knowledge of program semantics to optimize bidding strategies and execution plans jointly. For example, they employ generic checkpointing and/or migration techniques, but cannot extend plans with *sync* jobs intelligently as *Cumulon* does. Finally, support for auction-based markets has also been studied from the perspective of service providers [16, 20], which complements *Cumulon*'s focus on the perspective of users.

5 Conclusion

The increasing commoditization of computing, aided by the growing popularity of public clouds, holds the promise to make big-data analytics accessible to more users than ever before. However, in contrast to the relative ease of obtaining hardware resources, most users still find it difficult to use these resources effectively. In this paper, we have presented *Cumulon*, an ongoing project at Duke University aimed at simplifying—from users' perspective—both development and deployment of large-scale, matrix-based data analysis on public clouds. *Cumulon* is about providing the appropriate abstractions to users: it applies the philosophy of database systems—"*specify what you want, not how to do it*"—to the world of matrices, linear algebra, and cloud. As discussed in this paper, achieving this vision involves many challenges. *Cumulon* builds on existing cloud programming platforms, but carefully avoids the limitations of their models and dependency on specific platforms using a simple yet flexible storage and execution framework, which also allows database- and HPC-style processing to be incorporated for higher efficiency. *Cumulon* automatically makes cost-based decisions on a large number of issues, from the setting of execution parameters to the choice of clusters and bidding strategies. *Cumulon* also fully embraces uncertainty, not only for better performance modeling, but also in formulating its optimization problem to capture users' risk tolerance. The combination of these design choices and techniques enables *Cumulon* to provide an end-to-end, user-facing solution for running matrix-based workloads in the cloud.

Future Work As discussed in Section 3, we are actively investigating online optimization in *Cumulon*. While *Cumulon* alerts the user upon detecting significant deviations from the predicted execution progress or market price, it currently relies on the user to take further actions. We would like to make *Cumulon* adapt to changing markets, recover from mispredictions, and deal with unexpected events intelligently, with less user supervision.

Our work on *Cumulon* to date has mostly focused on the *Cumulon* backend; we are still working on devising friendly interfaces for programming, performance monitoring, and interactive optimization. Instead of a full-fledged language, we envision a high-level language targeting just the domain of linear algebra, similar in syntax to MATLAB and R. Users write matrix-based, compute-intensive parts of their analysis in this language. These parts are embedded in a program written in a host language (such as R) and preprocessed by *Cumulon*. Users specify their time, money, and risk tolerance requirements (as well as other pertinent information such as authentication information) in a deployment descriptor that go along with the program. We also plan to develop

rewrites explicitly.

a graphical user interface for *Cumulon*, to better present output from the optimizer and the performance monitor, and to make input more user-friendly.

Accurate performance prediction for arbitrary matrix computation is challenging. Many analysis techniques are iterative, and their convergence depends on factors ranging from data characteristics (e.g., condition numbers of matrices) to starting conditions (e.g., warm vs. cold). So far, *Cumulon* has taken a pragmatic approach—it focuses on optimizing and costing a single or fixed number of iterations. User need to supply the desired number of iterations, or reason with "rates" of progress and cost of iterative programs. Better techniques for predicting convergence will be useful, but it remains unclear whether we can bound the prediction uncertainty to low enough levels to meet users' risk tolerance requirements. Making the optimizer online may be our best option.

Cumulon currently only exploits intra-job parallelism. However, there are also opportunities for inter-job parallelism in statistical analysis workloads [14]. It would be nice to support both forms of parallelism. *SystemML* has made promising progress in this direction [2], but we will need to extend the optimization techniques to consider cluster provisioning and bidding strategies in this setting.

To keep up with rapid advances in big-data analysis, we must make *Cumulon* an extensible and evolvable platform. Developers should be able to contribute new computational primitives to *Cumulon* in a way that extends not only its functionality but also its optimizability. While implementing a new operator may be easy, telling *Cumulon* how to optimize the use of this new operator is much harder. *Cumulon* relies on high-quality cost models that come with assessment of prediction uncertainty, but we cannot always assume such models to be available from the start; instead, we need techniques for automatically building reasonable "starter" models as well as identifying and improving poor models, in order to provide meaningful risk assessment for users. In general, supporting extensibility in optimizability—from cost estimation to search algorithms—poses many challenges that have until now been relatively less studied.

Acknowledgments This work has been supported by NSF grants 0916027, 0917062, 0964560, 1320357, a 2010 HP Labs Innovation Research Award, and grants from Amazon Web Services.

References

[1] A. Andrzejak, D. Kondo, and S. Yi. Decision model for cloud computing under SLA constraints. In *Proceedings of the 2010 IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 257–266, Miami, Florida, USA, Aug. 2010.

[2] M. Boehm, S. Tatikonda, B. Reinwald, P. Sen, Y. Tian, D. Burdick, and S. Vaithyanathan. Hybrid parallelization strategies for large-scale machine learning in SystemML. *Proceedings of the VLDB Endowment*, 7(7):553–564, 2014.

[3] N. Chohan, C. Castillo, M. Spreitzer, M. Steinder, A. N. Tantawi, and C. Krintz. See spot run: Using spot instances for MapReduce workflows. In *Proceedings of the 2010 Workshop on Hot Topics on Cloud Computing*, Boston, Massachusetts, USA, June 2010.

[4] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. MAD skills: New analysis practices for big data. In *Proceedings of the 2009 International Conference on Very Large Data Bases*, pages 1481–1492, Lyon, France, Aug. 2009.

[5] S. Das, Y. Sismanis, K. S. Beyer, R. Gemulla2, P. J. Haas, and J. McPherson. Ricardo: Integrating R and Hadoop. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, Indianapolis, Indiana, USA, June 2010.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 2004 USENIX Symposium on Operating Systems Design and Implementation*, pages 137–150, San Francisco, California, USA, Dec. 2004.

[7] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative machine learning on MapReduce. In *Proceedings of the 2011 International Conference on Data Engineering*, Hannover, Germany, Apr. 2011.

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

[9] 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. *Proceedings of the VLDB Endowment*, 5(12):1700–1711, 2012.

[10] B. Huang, S. Babu, and J. Yang. Cumulon: Optimizing statistical data analysis in the cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, New York City, New York, USA, June 2013.

[11] B. Huang, N. W. D. Jarrett, S. Babu, S. Mukherjee, and J. Yang. Leveraging spot instances for cloud-based statistical data analysis. Technical report, Duke University, July 2014.

[12] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2007 EuroSys Conference*, pages 59–72, Lisbon, Portugal, Mar. 2007.

[13] K. Kambatla, A. Pathak, and H. Pucha. Towards optimizing hadoop provisioning for the cloud. In *Proceedings of the 2009 Workshop on Hot Topics on Cloud Computing*, Boston, Massachusetts, USA, June 2009.

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

[15] H. Liu. Cutting MapReduce cost with spot market. In *Proceedings of the 2011 Workshop on Hot Topics on Cloud Computing*, Portland, Oregon, USA, June 2011.

[16] M. Mazzucco and M. Dumas. Achieving performance and availability guarantees with spot instances. In *Proceedings of the 2011 IEEE International Conference on High Performance Computing and Communication*, pages 296–303, Banff, Alberta, Canada, Sept. 2011.

[17] V. Menon and K. Pingali. High-level semantic optimization of numerical codes. In *Proceedings of the 1999 ACM/IEEE Supercomputing Conference*, pages 434–443, Rhodes, Greece, June 1999.

[18] V. Rokhlin, A. Szlam, and M. Tygert. A randomized algorithm for principal component analysis. *SIAM Journal on Matrix Analysis and Applications*, 31(3):1100–1124, Aug. 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. In *Proceedings of the 2010 IEEE International Conference on Cloud Computing Technology and Science*, pages 721–726, Indianapolis, Indiana, USA, Nov. 2010.

[20] Y. Song, M. Zafer, and K.-W. Lee. Optimal bidding in spot instance market. In *Proceedings of the 2012 IEEE International Conference on Computer Communications*, pages 190–198, Orlando, Florida, USA, Mar. 2012.

[21] E. R. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. E. Gonzalez, M. J. Franklin, M. I. Jordan, and T. Kraska. MLI: An API for distributed machine learning. In *Proceedings of the 2013 International Conference on Data Mining*, pages 1187–1192, Dallas, Texas, USA, Dec. 2013.

[22] S. Tang, J. Yuan, and X.-Y. Li. Towards optimal bidding strategy for Amazon EC2 cloud spot instance. In *Proceedings of the 2012 IEEE International Conference on Cloud Computing*, number 91–98, Honolulu, Hawaii, USA, June 2012.

[23] A. Verma, L. Cherkasova, and R. H. Campbell. ARIA: automatic resource inference and allocation for MapReduce environments. In *Proceedings of the 2011 International Conference on Autonomic Computing*, Karlsruhe, Germany, June 2011.

[24] W. Voorsluys and R. Buyya. Reliable provisioning of spot instances for compute-intensive applications. In *Proceedings of the 2012 IEEE International Conference on Advanced Information Networking and Applications*, pages 542–549, Fukuoka, Japan, Mar. 2012.

[25] S. Yi, A. Andrzejak, and D. Kondo. Monetary cost-aware checkpointing and migration on Amazon cloud spot instances. *IEEE Transactions on Services Computing*, 5(4):512–524, 2012.

[26] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2010 Workshop on Hot Topics on Cloud Computing*, Boston, Massachusetts, USA, June 2010.

[27] Y. Zhang, H. Herodotou, and J. Yang. RIOT: I/O-efficient numerical computing without SQL. In *Proceedings of the 2009 Conference on Innovative Data Systems Research*, Asilomar, California, USA, Jan. 2009.

[28] Y. Zhang and J. Yang. Optimizing I/O for big array analytics. *Proceedings of the VLDB Endowment*, 5(8):764–775, June 2012.

[29] Z. Zhang, L. Cherkasova, A. Verma, and B. T. Loo. Meeting service level objectives of Pig programs. In *Proceedings of the 2012 International Workshop on Cloud Computing Platforms*, pages 8:1–8:6, Bern, Switzerland, Apr. 2012.