Cümülön: Matrix-Based Data Analytics in the Cloud with Spot Instances

Botong Huang,[1] Nicholas W.D. Jarrett,[2] Shivnath Babu,[3] Sayan Mukherjee,[4] Jun Yang[5]
Duke University
Department of Computer Science[1,3,5] and Department of Statistical Science[2,4]
bhuang@cs.duke.edu, nwj2@stat.duke.edu, shivnath@cs.duke.edu, sayan@stat.duke.edu, junyang@cs.duke.edu

ABSTRACT

We describe Cümülön, a system aimed at helping users develop and deploy matrix-based data analysis programs in a public cloud. A key feature of Cümülön is its end-to-end support for the so-called spot instances*—machines whose market price fluctuates over time but is usually much lower than the regular fixed price. A user sets a bid price when acquiring spot instances, and loses them as soon as the market price exceeds the bid price. While spot instances can potentially save cost, they are difficult to use effectively, and run the risk of not finishing work while costing more. Cümülön provides a highly elastic computation and storage engine on top of spot instances, and offers automatic cost-based optimization of execution, deployment, and bidding strategies. Cümülön further quantifies how the uncertainty in the market price translates into the cost uncertainty of its recommendations, and allows users to specify their risk tolerance as an optimization constraint.*

1 Introduction

Publicly available *clouds*, such as Amazon EC2, Microsoft Azure, and Google Cloud, have made it easier to acquire computing resources on demand. Typically, users rent machines in the cloud at a fixed rate (e.g., $0.145 per hour for an Amazon `c1.medium`-type machine). These machines can be used with no interruption until users release them. With such fixed-price rentals, however, a cloud provider may find its clusters underutilized and unable to generate revenue from vacant machines. On the other hand, when the demand is high, there is no way to fulfill all user requests even if some users may be willing to pay higher prices. One pricing policy that seems to be gaining traction in recent years is to allow users to bid for machines, whose prices fluctuate according to supply and demand; hence, resources will be allocated to those who need them the most, and the cloud provider can increase its profit.

Amazon EC2 is a leader with this pricing policy. In addition to fixed-price *on-demand instances* (machines), Amazon EC2 also offers *spot instances*, whose market price changes over time, but is usually significantly lower than the fixed price of the on-demand instances. A user acquires spot instances by setting a bid price higher than the current market price, and pays for them based on the changing market price. However, as soon as the market price exceeds the bid price, the user will lose the spot instances.

From a user's perspective, spot instances offer a good cost-saving opportunity, but there are a number of difficulties in using them:

- Work done on spot instances will be lost when they are reclaimed. There is no guarantee when this event will happen, and when it happens, there is little time to react. In this sense, the loss of spot instances is similar to machine failures, but it is one of the worst kinds possible—it amounts to a massive correlated failure where *all* spot instances acquired at the (now exceeded) bid price are lost at the same time. The user still has to pay for their use before the point of loss.[1] What can we do to mitigate such risks? Should we checkpoint execution progress on the spot instances? More checkpointing lowers the cost of recovery in the event of loss, but it also increases execution time and cost, as well as the possibility of encountering a loss. What is worth checkpointing and where do we save it?
- There are also numerous options to consider for bidding. How many spot instances should we bid for? More machines potentially imply faster completion and hence lower chance of loss during execution, but a large parallelization factor often leads to lower efficiency and higher overall monetary cost, not to mention the possibility of paying for lots of spot instances without getting any useful work out of them. Moreover, how do we set the bid price? Bidding high decreases the chance of loss, but increases the average price we expect to pay over time. Does it make sense to bid above the fixed price of on-demand instances?
- When working with spot instances, we face a great deal of uncertainty, a primary source of which is the variability of market prices. From a cloud provider's perspective, a good average-case behavior may be enough to make a decision, but from a user's perspective, cost variability is a major concern. How do we quantify the amount of uncertainty incurred by the use of spot instances? Given a user's risk tolerance and cost constraints, does it even make sense to use them? Is there any way to bound this uncertainty while still saving some cost in the expected sense?

Our Contributions We present our answers to the challenges above in the context of a system called *Cümülön* (i.e., *Cumulon* with spots). Cumulon [5, 6] is a system aimed at simplifying the development and deployment of statistical analysis of big data on public clouds. With Cumulon, users write high-level programs using matrices and linear algebra, without worrying about how to

[1] Amazon EC2's pricing scheme is actually more nuanced and can lead to some rather interesting bidding strategies; more details are in Section 4.2.

map data and computation to underlying cloud computing platforms. Given a program, Cumulon automatically optimizes its implementation alternatives, execution parameters, as well as hardware provisioning and configuration settings. In this paper, we describe how we introduce end-to-end support for spot instances in Cümülön. Starting with an optimized "baseline" plan using fixed-price instances, Cümülön makes intelligent decisions on whether and how to bid for additional spot instances, and how to use them effectively to reduce expected cost while staying within users' risk tolerance. We show how to build a highly elastic computation and storage engine for matrices on top of spot instances.

A key desideratum of Cümülön is letting users specify their objectives and constraints in straightforward terms. Given an optimized baseline plan using only fixed-price instances, a user can ask Cümülön to find the plan involving spot instances that minimizes the expected monetary cost, while satisfying the constraint that the actual cost is within $(1+\delta)$ of the baseline cost with probability no less than σ. By tuning δ and σ, the user tells Cümülön how much risk she is willing to take while trying to reduce the expected cost.

Example 1 (RSVD-1 Plan Space)**.** *Consider* singular value decomposition *(*SVD*) of matrices, which is fundamental to many applications of statistical data analysis. In [13], the first and most expensive step of the randomized SVD algorithm, 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* $\mathbf{G} \times (\mathbf{A} \times \mathbf{A}^{\intercal})^k \times \mathbf{A}$.

The baseline plan (optimal without spot instances) of RSVD-1 (with l = 2,048*,* m = n = 102,400*, and* k = 5*), picked by Cumulon, costs* \$5.09 *and runs under* 11.7 *hours, using* 3 *machines of type* `c1.medium` *at the fixed price of* \$0.145 *per hour. Under the user-specified risk tolerance of* $\delta = 0.05$ *and* $\sigma = 0.9$*, Cümülön is able to recommend that the user bid for additional* 77 `c1.medium` *spot instances at* \$0.10 *each per hour (versus the current market price of* \$0.02*. This plan reduces the expected overall cost to* \$3.78 *(*26*% improvement) while staying within the risk tolerance.*

To further help the user understand the (sometimes non-obvious) trade-off between bidding strategies, Cümülön can produce a visualization such as Figure 1a. From this figure, we see that bidding for more machines generally decreases expected cost—as this problem is large in size and amenable to parallelization. Interestingly, bidding at higher prices also tends to lower expected cost in this case. In general, the higher we bid, the longer we expect to hold spot instances for useful work, though we pay higher average price over time. Here, the advantage outweighs the disadvantage. It is reasonable to bid even above *the fixed price (*\$0.145*), since the average market price we end up paying over time can still be lower.*

The upper-right region of Figure 1a is where no plan with given bid price and number of spot instances meets the user-specified risk tolerance. Intuitively, the combination of high bid price and large number of spot instances increases the risk of overshooting the baseline cost.

If the user is willing to take a higher risk, she can lower σ *from* 0.9 *to* 0.8*. Cümülön will then revise the figure to Figure 1b. We see that the region of valid bidding strategies expands, and the expected cost can be further lowered to* \$3.73*.*

Cümülön does a considerable amount of work behind the scenes to help users make high-level decisions without worrying about low-level details. In terms of system support for spot instances, Cümülön uses a dual-store design consisting of both *primary nodes* (regular, fixed-price instances) and *transient nodes* (variable-price spot instances). The *primary store* leverages the reliability of primary nodes to offer persistent storage without relying on separate cloud-based storage services that are costly and often inefficient. The *transient store* provides elastic, fast storage for transient nodes without overwhelming the primary store. To combat loss of progress when transient nodes are reclaimed, Cümülön uses a combination of (implicit) caching and (explicit) *sync* operations to copy selected state from the transient store to the primary one. Taking advantage of declarative program specification (with matrices and linear algebra), Cümülön uses fine-grained lineage information to minimize the work required to recover lost data.

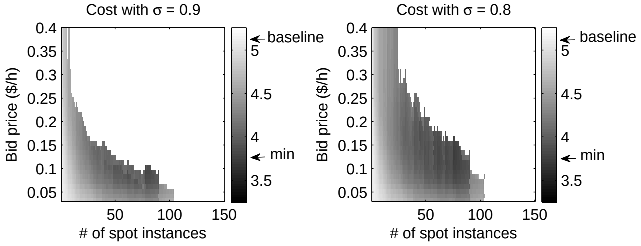


Figure 1: Estimated expected cost of the optimal plans for RSVD-1, as we vary the bid price and the number of spot instances we bid for. The cost is shown using intensity, with darker shades indicating lower costs. In the upper-right region, plans fail to meet the user-specified risk tolerance; hence, this region shows the baseline cost without bidding.

In terms of optimization support for spot instances, Cümülön optimizes not only implementation alternatives, execution parameters, and configuration settings (as Cumulon does), but also bidding strategies and how to add *sync* operations during execution. To make intelligent decisions, Cümülön estimates execution times as well as the time needed for *sync* and recovery—which depends on how much data might be lost, how much of that will still be needed, and how much work is involved in recomputing the required portion. Furthermore, Cümülön has to reason with unknown future market prices, and in particular, predict when we will lose the spot instances. The market introduces a great deal of uncertainty; Cümülön quantifies how this uncertainty translates into the cost uncertainty of its recommendations, and considers how various options impact the resulting uncertainty.

While most part of this paper assumes a modified version of Amazon's pricing policy (see Section 4.2 for more details), our system and framework allow any pricing policy to be plugged in. This flexibility allows us to explore interesting policy questions, e.g., how would our bidding strategy change if we have to pay the bidding price instead of the changing market price? In the remainder of this paper, we describe the inner workings of Cümülön, evaluate how well Cümülön supports the use of spot instance, and explore how the pricing policy affects bidding strategies.

2 Background on Cumulon

This section gives a brief review of Cumulon, which supported only regular, fixed-price nodes. For more details, please see [5].

Storage Since we target matrix workloads, Cumulon provides a *distributed tile store* for matrices. Matrices are accessed in the unit of *tiles*, which are submatrices of fixed (but configurable) size. Within each tile, elements are stored in either row-major or column-major order, using either sparse or dense format. Our execution model guarantees that a tile has either multiple concurrent readers, or one writer (producer); it is never updated in-place. Currently, Cumulon implements its tile store on top of a HDFS.

Execution Cumulon executes an input program as a sequence of *jobs* specified by a *physical plan template* $Q = \{q_1, q_2, \ldots, q_m\}$,

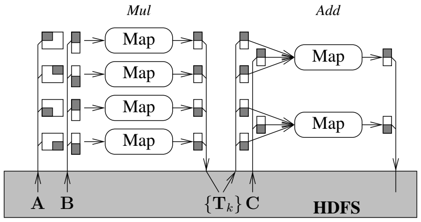


Figure 2: (From [5]) Cumulon's map-only Hadoop jobs for $\mathbf{A} \times \mathbf{B} + \mathbf{C}$. Shaded blocks represent submatrices. $\{\mathbf{T_k}\}$ is the collection of results of from submatrix multiplies, yet to be grouped and summed.

where q_j is the physical plan template for job j and is represented as a DAG of *physical operators*. Cumulon executes each job using multiple parallel and independent *tasks* that do not communicate with each other. Each task of job j runs an instance of q_j: it produces an *output split* (a portion of the job output, disjoint from the other tasks) using an *input split* (portions of the job input, possibly overlapping with the other tasks). Cumulon executes a task 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—we pass tiles instead of elements, to reduce overhead, and to enable the use of highly tuned BLAS library on submatrices.

Each node in the cluster is configured into several *slots*, each of which can execute one task at a time. Cumulon *scheduler* assigns tasks to slots. A job with more tasks than slots will take multiple waves to finish. The next job does not start until all tasks in the current job is done. Data are passed between jobs only through the distributed tile store. Currently, Cumulon's execution engine is built on top of Hadoop, but in away that does not follow the standard MapReduce model. Each Cumulon job runs as a map-only Hadoop job. However, different tasks can read overlapping portions of the input data, directly from the distributed tile store when needed; there is no requirement of disjoint input partitioning or shuffle-based data passing as in MapReduce. For example, Figure 2 illustrates how Cumulon implements $\mathbf{A} \times \mathbf{B} + \mathbf{C}$; the summation step of × here has been folded into the job that adds $\mathbf{C}$. In [5], we have demonstrated that this execution model enables far more efficient support for matrix workloads than approaches based on MapReduce.

Optimization Given a program and input data characteristics (e.g., matrix dimensions and sparsity), Cumulon applies rewrite rules (e.g., linear algebra equivalences) to obtain a physical plan template. Then, using a cost-based optimizer, Cumulon chooses hardware provisioning settings (e.g., number and type of nodes to use), system configuration settings (e.g., number of slots per node), and execution parameters (e.g., splits for each job, or other physical operators parameters). The optimizer looks for the plan with the lowest expected monetary cost among those expected to complete by a user-specified deadline.

To enable cost-based optimization, Cumulon predicts task completion time using models for individual physical operators trained using benchmarks on each machine type; more details about this model will be reviewed later, when we discuss how to extend it for Cümülön in Section 4.3. Then, to predict job completion time, Cumulon simulates the behavior of its scheduler; this approach accounts for variance in task completion times as well as potentially heterogeneous clusters.

Remark Currently, both Cumulon and Cümülön are built on top of Hadoop/HDFS, although we note that they have been designed such that they could be implemented using alternative cloud computing platforms, such as *Spark* [22] and *Dryad* [9]. Most techniques presented in this paper are not specific to Hadoop/HDFS and can be applied readily to other platforms.

3 System Support for Transient Nodes

There are several desiderata in supporting transient nodes. First, we want to handle a large number of transient nodes (which can be much more than the number of primary nodes). Second, we want to allow their instant arrival and departure. In particular, in the event that the market price for a set of transient nodes exceeds their bid price—which we call a *hit* for brevity—we do not assume that there is enough time to checkpoint execution progress on these nodes before we lose them.

Cumulon's execution model is already elastic in nature. Tasks in a job are independent and can run anywhere in any order. Cümülön further extends Cumulon's scheduler to work for the general case where the cluster contains multiple machine types (e.g., `m1.small` vs. `c1.medium`), which may arise, for example, if we bid for a particular type of nodes whose current market price becomes low enough to make it cost-effective. The extension supports dynamic division of a job into tasks of various sizes, each tailored toward a specific machine type. We omit the details because support for heterogeneous clusters is not the focus of this paper.

What to do with data—specifically, the results produced by the tasks—is more challenging, because losing such data to an inopportune hit can lead to significant loss of work. In the remainder of this section, we show how to tackle this challenge using a dual-store design with caching and *sync*, and how to recover from a hit.

3.1 Dual-Store Design

Before describing our design, we briefly discuss several strawman solutions and why we ruled them out. **1)** Using a distributed *storage service offered by the cloud provider* (such as Amazon S3) is a simple way to make Cümülön tile store persistent, but storing all intermediate results in it is both inefficient and prohibitively expensive. **2)** Using a *single HDFS on top of both primary and transient nodes* is a natural extension to Cumulon that that fully exploits the local storage on the available nodes. However, HDFS is not designed to withstand massive correlated node failures, which happen with a hit. HDFS's decommission process simply cannot work fast enough to prevent data loss. **3)** Using a *single HDFS on primary nodes only* as the Cümülön tile store is cheap and reliable. Tasks on transients nodes would write their results to the primary nodes, which will be preserved in the event of a hit. However, since there are usually significantly more transient nodes than primary ones, these writes will easily overwhelm the primary nodes (as we will demonstrate with experiments in Section 5).

As mentioned in Section 1, Cümülön uses a dual-store design, where the primary nodes together form a *primary store*, and the transient nodes together form a separate *transient store*. We call the primary (transient, resp.) store the *home store* of a primary (transient, resp.) node. Sitting on top of the two stores, a distributed tile manager keeps track of where tiles are stored and mediates accesses to tiles. To process a ***read***, a node first attempts to find (a copy of) the requested tile in its home store. If not found, the node will fetch the tile from the non-home store, and then store a copy in the home store. To process a ***write***, a node simply writes the tile to its home store.

As a further optimization, we cache a tile in the transient store only if this tile will be read later.[2] To enable this optimization,

[2] Note that we always cache in the primary store, because even if a tile will not be read again in normal execution, it may be useful for recomputing other useful tiles during recovery (Section 3.3).

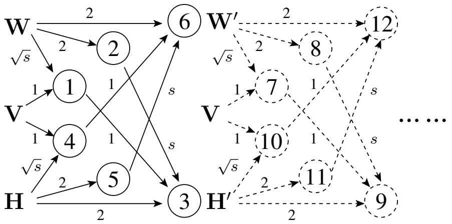


Figure 3: Dependencies among jobs for two iterations of GNMF. Edges are labeled with read factors, where s denotes the number of tasks per job.

Cümülön gives its tile manager an extra "cacheable" bit of information for each matrix that the program computes, based on the notion of *read factor* further described below. A tile of $\mathbf{A}$ will be cached by the transient store only if the "cacheable" bit for $\mathbf{A}$ is currently 1.

The *read factor* of a matrix $\mathbf{A}$ in job j, denoted by $\gamma_j^{\mathbf{A}}$, is the average number of times that each tile of $\mathbf{A}$ is read by job j. The read factor can be readily obtained at optimization time as a function of the split size by analyzing j's physical plan template. During a job i, Cümülön sets the "cacheable" bit for $\mathbf{A}$ to 1 iff $\sum_{j \geq i} \gamma_j^{\mathbf{A}} > 1$. To illustrate, consider the following example.

Example 2 (Read Factors in GNMF)**.** Gaussian non-negative matrix factorization *(*GNMF*) [10] has many applications including vision, document clustering, and recommendation systems. Each step of the iterative GNMF algorithm computes the following:*

$$\mathbf{H}' \leftarrow \mathbf{H} \circ (\mathbf{W}^{\intercal} \times \mathbf{V}) \oslash (\mathbf{W}^{\intercal} \times \mathbf{W} \times \mathbf{H});$$

$$\mathbf{W}' \leftarrow \mathbf{W} \circ (\mathbf{V} \times \mathbf{H}^{\intercal}) \oslash (\mathbf{W} \times \mathbf{H} \times \mathbf{H}^{\intercal}).$$

Here, $\circ$ denotes element-wise multiply and $\oslash$ denotes element-wise divide. $\mathbf{V}$ is a sparse $n \times m$ matrix; $\mathbf{W}$ and $\mathbf{H}$ are dense $n \times k$ and $k \times m$ (resp.) matrices, where k is much smaller than n and m. $\mathbf{W}'$ and $\mathbf{H}'$ become $\mathbf{W}$ and $\mathbf{H}$ (resp.) in the next iteration.

Figure 3 shows the dependencies among jobs and matrices for two GNMF iterations. Cümülön compiles one iteration into 6 *jobs. Job* 1 *computes $\mathbf{W}^{\intercal} \times \mathbf{V}$. It partitions $\mathbf{V}$ into a grid of $n' \times n'$ submatrices, and $\mathbf{W}$ into a column of n' submatrices; each task multiplies a pair of submatrices. Each submatrix of $\mathbf{W}$ is read n' times, so the read factor (of each tile in $\mathbf{W}$ in job* 1*) $\gamma_1^{\mathbf{W}} = n'$, or the square root of the number of tasks.*

Note that our storage layer does not aggressively push writes between the two stores; replication across stores happens through caching on reads. This policy does not eliminate the risk of losing data produced by the transient nodes after a hit. However, there are several advantages. **1)** This policy avoids write traffic jams from the transient store to the primary store, by piggybacking writes on subsequent reads and thereby spreading them out. **2)** This policy naturally gives priority to data with higher utility; the more often a tile is read, the more likely it will be cached at the primary store. **3)** Compared with aggressively pushing writes to the primary store, this policy can potentially save many such writes. Utility of intermediate results will decrease once jobs requiring them complete; if there is no hit by then, tiles produced and "consumed" by transient nodes themselves will not be written to the primary store.

Currently, Cümülön implements the primary and transient stores as separate HDFS on respective nodes. Conveniently, HDFS provides efficient shared storage for all nodes with the same home store; thus, a tile cached upon one node's read request will benefit all nodes in the same store. Cümülön uses a replication factor of 3 within each HDFS to guard against data loss due to occasional node failures (which could still happen even with primary nodes). However, as discussed earlier, we do not assume that the transient store can preserve any data when the cluster is hit.

3.2 Sync Jobs

Caching data on read is opportunistic and not enough to bound data loss in the event of a hit. Losing a tile, especially when late in execution, will trigger expensive recomputation going back all the way to the beginning of execution unless sufficient intermediate results survive in the primary store. Thus, we introduce explicit *sync* jobs to ensure that a set of matrices completely "persist" on the primary store. To a persist matrix $\mathbf{A}$, Cümülön consults its tile manager to identify the collection of $\mathbf{A}$ tiles present *only* in the transient store. Then, the *sync* job reads these tiles (from the transient store) and writes them to the primary store. As with other jobs, this job executes as multiple parallel and independent tasks, each responsible for a subset of the tiles.

Hardwiring a rigid syncing strategy into the system (e.g., syncing all intermediate results periodically or even after every non-*sync* job) is suboptimal, as the best strategy depends on many factors: when and how often an intermediate result matrix $\mathbf{A}$ will be used later, how likely a hit will occur before the uses of $\mathbf{A}$, how much of $\mathbf{A}$ will be cached by the primary store over time, and how expensive it is to recompute the part of $\mathbf{A}$ required when recovering from a hit. Some of these factors can be determined by statistic program analysis; some further depend on future market prices. It would unrealistic to expect users to come up good syncing strategies manually. Therefore, Cümülön considers the choice of a syncing strategy as an integral part of its optimization (Section 4).

3.3 Recovering from a Hit

In practice, syncing all intermediate results is too expensive, and even if we do, a hit may still occur during a *sync* job. Cümülön supports fine-grained data-driven recovery: it performs only the computation needed to recover the specific portions of data that are missing and required for resuming execution. Cümülön does not rely on having any complete snapshot of the execution state. Thanks to its knowledge of the physical plan template, Cümülön is able to redo a job "partially," using tasks that may differ from those in the original execution.

To help determine the dependencies between input and output data, each physical operator in Cümülön supports a *lineage function*, which returns the subset of input tiles required for computing a given subset of output tiles. To obtain the lineage function of a job, we simply compose the lineage functions for the physical operators in the job's physical plan template. Suppose a job reads matrix $\mathbf{A}$ and produces matrix $\mathbf{B}$. We use $\Lambda_{\mathbf{B}}^{\mathbf{A}}(\mathcal{B})$ to denote the subset of $\mathbf{A}$ tiles required for computing the given subset $\mathcal{B}$ of $\mathbf{B}$ tiles, as determined by the lineage function of the job.

Once Cümülön detects a hit—say during the execution of job j_{hit}—the scheduler stops issuing new tasks for j_{hit}, and gives a short time for ongoing tasks on the primary nodes to either complete or fail (due to missing data or time running out). Next, Cümülön calculates, for each job up to j_{hit} and each matrix $\mathbf{A}$ produced by these jobs, the set $X^{\mathbf{A}}$ of *deficient tiles* in $\mathbf{A}$, i.e., those that must be recomputed in order to resume execution. This calculation starts with job j_{hit} and works backwards. Consider $\mathbf{A}$ produced by job j. Let $P^{\mathbf{A}}$ denote the set of *persisted tiles* in $\mathbf{A}$, i.e., those currently available in the primary store. If $\mathbf{A}$ will be read by any job following j_{hit}, then all non-persisted tiles are deficient. Otherwise, let $\mathcal{B}$ denote the set of matrices produced by any job among $j+1, j+2, \ldots, j_{\text{hit}}$ that reads $\mathbf{A}$. To recover the deficient tiles for each $\mathbf{B} \in \mathcal{B}$, we need the subset $\Lambda_{\mathbf{B}}^{\mathbf{A}}(X^{\mathbf{B}})$ of $\mathbf{A}$'s tiles. Therefore, $X^{\mathbf{A}} = \bigcup_{\mathbf{B} \in \mathcal{B}} \Lambda_{\mathbf{B}}^{\mathbf{A}}(X^{\mathbf{B}}) \setminus P^{\mathbf{A}}$.

Then, Cümülön runs a series of "partial" jobs, one for each job up to j_{hit} with any deficient output tile. Each partial job j has the same physical plan template as the original job j, but runs only on the primary nodes and produces only the deficient tiles as determined above. After these partial jobs complete, execution can resume from job $j_{\mathsf{hit}} + 1$.

Note that the division of work in a partial job into tasks can be quite different from the original execution; in other words, Cümülön performs recovery in a more flexible, data-driven manner than just redoing a subset of the original tasks. Furthermore, Cümülön does not track lineage explicitly, but instead infer it as needed from the physical plan template. Such features are possible because Cümülön programs are specified declaratively using a vocabulary of operators with known semantics. These features distinguish Cümülön from other systems with lineage-based recovery (such as *Spark* [21]) that need to support black-box computation.

4 Optimization

There is a huge space of alternatives for running a Cümülön program with transient nodes—from execution to bidding to syncing. Moreover, Cümülön seeks to quantify the uncertainty in the costs of its recommendations, and allows users to specify their risk tolerance as an optimization constraint. We impose several restrictions on the plan space, either to keep optimization tractable or to simplify presentation. Then, we discuss how to extend our solution for the simplified optimization problem to consider more complex plans. Specifically, we start with the following restrictions:

(Augmenting a Baseline Plan) Given a program to optimize, we begin with a *baseline plan* with n_{prim} primary nodes of a specific machine type at the fixed price of p_{prim} (per machine per unit time), and no transient nodes. The program's physical plan template Q consists of a sequence of m jobs; we denote by q_j the physical plan template of job j, where $1 \leq j \leq m$. This baseline plan (involving no transient nodes) can be the lowest-cost plan found by Cumulon, under a user-specified completion time constraint.

We only consider plans that *augment* this baseline plan with transient nodes: 1) we will not change the set of primary nodes; 2) we will not change Q, except for adding *sync* jobs.

The baseline plan makes it easy for users to specify risk tolerance. Suppose the estimated cost of the baseline plan is c. Then, the risk tolerance constraint can be specified as (δ, σ), which means that we only consider plans whose costs are within $(1 + \delta)c$ with probability no less than σ. Note that by bounding the plan cost, this constraint also places a soft upper bound on the completion time of the plan (because cost increases with time). Without the help of the baseline plan, it would be much more difficult for users to come up with appropriate constraints.

(Optimizing on Start) We assume that we make our optimization decision at the start of the program. The decision consists of two parts, bidding and syncing.

(Bidding for a Homogeneous Cluster) We assume that we only bid for transient nodes of the same machine type as the primary ones. Suppose the current market price for this type of transient nodes is p_0. The bidding strategy is characterized by $(\hat{p}, n_{\mathsf{tran}})$, where $\hat{p} \geq p_0$ is the bid price and $n_{\mathsf{tran}} \geq 0$ is the number of transient nodes to bid for.

As discussed in Section 3, Cümülön in fact has full system support for heterogeneous clusters. However, optimization for heterogeneous clusters is considerably more complex; we are currently still working on refining the cost models for heterogeneous clusters.

(Syncing after Output) We assume that we sync the output of job j only immediately after job j; in other words, we do not consider waiting to sync later. Thus, the syncing strategy is characterized by a mapping S from jobs to subsets of matrices they produce; $S(j)$ specifies the subset of matrices output by job j to be persisted in the primary store after job j completes. If $S(j) = \varnothing$, we move on to job $j + 1$ immediately after job j completes.

(Optimizing for One Bid) We make our current optimization decision based on the assumption that, if the cluster is hit, we will carry out remaining work using only the primary nodes. Under this assumption, the program execution can be divided into three phases:

- Until it is hit, the cluster executes the program at full strength with both primary and transient nodes. We call this phase the *pre-hit phase*, and denote its duration by T_{hit}.
- Upon being hit, if the execution has not finished, we enter the *recovery phase*, where the primary nodes perform recovery and complete the last non-*sync* job that started before the hit. We denote the duration of this phase by T_{rec}.
- Finally, the primary nodes complete any remaining (non-*sync*) jobs in the program. We call this phase the *wrap-up phase* and denote its duration by T_{rap}.

In sum, we solve the following optimization problem:

Given a baseline plan with estimated cost of c, physical plan template Q, and n_{prim} primary nodes, find bidding strategy $(\hat{p}, n_{\mathsf{tran}})$ and syncing strategy S that minimize the expected cost of three-phase execution, subject to the constraint that the actual execution cost is no more than $(1 + \delta)c$ with probability no less than σ.

This problem formulation implies that our optimization decision is myopic in the sense that it does not consider the future possibilities of bidding for additional transient nodes, voluntarily releasing transient nodes before completion, or dynamically re-optimizing the execution plan and syncing strategy, etc. In practice, however, we can re-run the optimization later and bid for a new set of transient nodes. We shall come back to such extensions in Section 4.7.

For the remainder of this section, we begin with the market price model in Section 4.1 and pricing scheme in in Section 4.2, which let us estimate T_{hit}, and calculate costs given cluster composition and lengths of the execution phases. We present models for estimating execution times in Sections 4.3 and 4.4. We then show how to combine these models to obtain the total cost distribution (with uncertainty) in Section 4.5, and how to solve the optimization problem in Section 4.6.

4.1 Market Price Model

In order to estimate T_{hit} and the cost of transient nodes, we need a model for predicting the future market price $p(t)$ at time t given the current market price $p(0) = p_0$. Cümülön allows any stochastic model to be plugged in, provided that it can efficiently simulate the stochastic process.

Our current market price model employs 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 diurnal and weekly periodicity, we wrap the time dimension in one-day and one-week cycles. Second, we model price spikes and their inter-arrival times as being conditionally independent given current and historical prices. We train the two components using historical spot price traces published by Amazon (details later in Section 5.3), and then combine the components to create a sample path. We omit further details because market price modeling is not the focus of this paper.

Although our full model captures periodicity, in this paper we instead use a simpler, non-periodic price model in our experiments for better interpretability of results. The reason is that with periodicity, costs and optimal strategies would depend also on the specific time when we start to run the program, making it harder for experiments to cover all cases and for readers to understand the impact of other factors. The technical report version of this paper [7] contains additional experiments showing how Cümülön is able to use the full periodic model to guide its decisions.

Given the market price model, current market price p_0, and bid price $\hat{p}$, we can repeatedly simulate the process to obtain multiple market price traces, stopping each one as soon as it exceeds $\hat{p}$. From these traces, we readily obtain the distribution of T_{hit}. For example, the top part of Figure 5 (ignore the bottom for now) shows the distribution of T_{hit} given $p_0 = \$0.02$ and $\hat{p} = \$0.2$, computed from our (non-periodic) model. We plot both PDF and CDF. From the figure, we see that the distribution roughly resembles the Lévy distribution, which characterizes the hitting time of a random walk. The PDF peaks shortly after the start, but has a long tail. If we are "lucky," we get to finish the program with a full-strength cluster (i.e., with both primary and transient nodes) without being hit. For example, say that in this case full-strength execution take 2 hours. We can then infer from the CDF in the figure that we get "lucky" with probability $1 - P(T_{\text{hit}} \leq 2h) = 0.32$.

4.2 Pricing Scheme

A *pricing scheme* computes the monetary cost of running a cluster given the fixed price p_{prim} of primary nodes and the time-varying market price $p(t)$ of transient nodes. Unless otherwise noted, we assume the following pricing scheme. Given n_{prim} primary nodes and n_{tran} transient nodes, and the lengths of the three execution phases (T_{hit}, T_{rec}, and T_{rap}), the total cost is

$$\begin{aligned} &C(n_{\text{prim}}, n_{\text{tran}}, T_{\text{hit}}, T_{\text{rec}}, T_{\text{rap}}) \\ &= n_{\text{prim}} p_{\text{prim}} (T_{\text{hit}} + T_{\text{rec}} + T_{\text{rap}}) + n_{\text{tran}} \int_0^{T_{\text{hit}}} p(t)\, dt. \end{aligned} \tag{1}$$

Basically, the primary nodes are charged at the constant rate of p_{prim} through the entire execution, while the transient nodes, working only during the pre-hit phase, are charged at the time-varying market price. For simplicity, we omit the cost of data ingress/egress at the beginning/end of the execution (e.g., from/to Amazon S3), because it is independent of our optimization decisions.

As hinted earlier (Footnote 1), Amazon EC2 actually uses a different pricing scheme. It rounds usage time to full hours, and for spot instances, it does not charge for the last partial hour of usage if they are lost. This policy is Amazon-specific and can lead to some rather interesting strategies, e.g., bidding low and intentionally holding transient nodes after completing work in hope that they will be reclaimed, making the last hour free. To make our results less specific to Amazon and easier to interpret, we consider fractional hours in computing costs by default in this paper (as Microsoft Azure and Google Cloud do). Cümülön can support the Amazon scheme (or any other alternative) if needed. In fact, in Section 5.4, we will investigate how optimal strategies change as we switch to the Amazon scheme, or another plausible pricing scheme where transients node are charged at their bidding price.

4.3 Job Time Estimation

Our goal here is to derive a function for estimating the execution time for a job. We build on the Cumulon job time estimator (for details see [5]). Briefly, Cumulon estimates job execution time from the execution time of its constituent tasks. The task execution time is broken down into two components: computation and I/O. The computation time is obtained from models for individual operators trained using micro-benchmarks. The I/O time model is trained as a function of the total amount of I/O and the cluster size, with the assumption that the sources and destinations of I/O requests are independently and uniformly distributed across the cluster. While a strong assumption, it worked quite well for linear algebra workloads on a cluster where all nodes participate in the distributed store and are expected to be available throughout the execution.

Cümülön continues to use the same computation time models as Cumulon, but it must extend the I/O time model in two situations where the uniformity assumption is clearly violated. 1) ***Egress from primary:*** Suppose a matrix $\mathbf{A}$ initially resides only on the primary store, and it is the first time that $\mathbf{A}$ is read by a job running on both primary and transient nodes. Here, all read requests target the primary store, and its read bandwidth may be the limiting factor. 2) ***Ingress to primary:*** Suppose a matrix $\mathbf{A}$ was produced by primary and transient nodes, and a *sync* job needs to ensure that the primary store has a complete version of $\mathbf{A}$. Here, all write requests target the primary store, and its write bandwidth may become the main constraint.

To account for these I/O patterns, Cümülön extends the I/O time model for the two cases above with two additional terms: both have the form of *total I/O amount / primary store bandwidth*; one is for data egress (reads) while the other is for data ingress (writes). Cümülön adds a weighted sum of these two terms to the I/O time predicted by Cumulon; we train the weights and measure the per-node bandwidths for each machine type using micro-benchmarks. Except for the two I/O patterns above, Cümülön uses the same I/O time estimate as Cumulon, because in those cases the program runs either on the primary nodes alone, or on both primary and transient nodes with uniformly distributed workload and data.

For a job with physical plan template q running on n_{prim} primary nodes and n_{tran} transient nodes, let $\mathfrak{T}(n_{\text{prim}}, n_{\text{tran}}, q)$ denote its estimated completion time, obtained using the method above. The extended I/O time model is invoked 1) if $n_{\text{tran}} > 0$ and the job reads a matrix residing entirely on the primary store, or 2) if the job is a *sync*. The context is always clear when the Cümülön optimizer performs this estimation.

4.4 Sync and Recovery Time Estimation

While Section 4.3 has laid the foundation for estimating job time, for a *sync* or recovery job, we still have to know, respectively, the fraction of data needed to be preserved or the fraction of the work needed to be redone. Recall from Sections 3.2 and 3.3 that at run time, Cümülön uses tile-level persistency and lineage information to determine precisely what data to preserve and what work to redo, but we do not have all information at optimization time. To enable estimation, Cümülön makes strong independence and uniformity assumptions. While these assumptions certainly do not hold in general, they work well for Cümülön because its focus on linear algebra workloads, which are much more predictable than arbitrary, black-box programs. We now discuss the two estimation problems.

Forward Propagation of Persistency We say that a job is ϕ-*completed* if it has run for a fraction ϕ of its total expected completion time. We assume that if a job producing matrix $\mathbf{A}$ is ϕ-completed, it will have produced the same fraction (ϕ) of $\mathbf{A}$'s tiles.

Let $\rho^{\mathbf{A}}_{j,\phi}$, the *persistency* of $\mathbf{A}$, denote the estimated fraction of $\mathbf{A}$ in the primary store at the time when job j is ϕ-completed, assuming that the cluster has been running at full strength since the beginning of $\mathbf{A}$'s production. Let $\gamma = \frac{n_{\text{prim}}}{n_{\text{prim}} + n_{\text{tran}}}$. Suppose $\mathbf{A}$ is produced by job j_0; we have $\rho^{\mathbf{A}}_{j_0,\phi} = \phi\gamma$, because each tile of $\mathbf{A}$ has a probability γ of being produced on the primary store.

We now estimate how the persistency of $\mathbf{A}$ changes as execution progresses in a full-strength cluster. Recall from Section 3.1 that $\gamma_j^{\mathbf{A}}$ denotes the read factor of $\mathbf{A}$ in job j, which can be obtained at optimization time by analyzing j's physical plan template. We calculate $\rho_{j,\phi}^{\mathbf{A}}$ from $\rho_{j-1,1}^{\mathbf{A}}$ as follows. Because Cümülön caches reads in its home store (Section 3.1), a read of $\mathbf{A}$ by a primary node can potentially increase the persistency of $\mathbf{A}$. For a tile to absent from the primary store when job j is ϕ-completed, the tile must be absent before j (which happens with probability $1 - \rho_{j-1,1}^{\mathbf{A}}$), and none of the $\phi\gamma_j^{\mathbf{A}}$ reads comes from a primary node (each of which happens with probability $1-\gamma$). Therefore, $\rho_{j,\phi}^{\mathbf{A}} = 1-(1-\rho_{j-1,1}^{\mathbf{A}})(1-\gamma)^{\phi\gamma_j^{\mathbf{A}}}$.

We estimate the time to *sync* a set $\mathcal{A}$ of matrices after job j as

$$\widetilde{T_{\mathit{sync}}^{\mathcal{A},j}} = \mathfrak{T}\Big(n_{\mathsf{prim}}, n_{\mathsf{tran}}, \mathit{sync}\big(\sum_{\mathbf{A}\in\mathcal{A}} (1-\rho_{j,1}^{\mathbf{A}}) \cdot \mathrm{size}(\mathbf{A})\big)\Big), \qquad (2)$$

where $\mathfrak{T}(n_{\mathsf{prim}}, n_{\mathsf{tran}}, \mathit{sync}(v))$ estimates the time it takes for a *sync* job to persist v amount of data on the transient nodes to the primary nodes, using the I/O time model of Section 4.3. Of course, if we *sync* $\mathbf{A}$ after job j, the persistency of $\mathbf{A}$ becomes 1 if the *sync* job completes, or $(1-\phi)\rho_{j,1}^{\mathbf{A}} + \phi$ if the *sync* job is ϕ-completed.

Backward Propagation of Deficiency Suppose the cluster is hit when job j_{hit} is ϕ-completed. Recall from Section 3.3 that the amount of recovery work depends on the number of deficient tiles in each matrix. Thus, our goal is to estimate, for each matrix $\mathbf{A}$ produced by jobs up to j_{hit}, the fraction of $\mathbf{A}$'s tiles that are deficient. We call this quantity the *deficiency* of $\mathbf{A}$, denoted by $\chi^{\mathbf{A}}$.

To this end, we introduce the *coverage* function, derived from the lineage function in Section 3.3. Suppose the job producing matrix $\mathbf{B}$ uses matrix $\mathbf{A}$ as input. Let $\lambda_{\mathbf{B}}^{\mathbf{A}}(f)$ return the estimated fraction of $\mathbf{A}$ required to compute the given fraction f of $\mathbf{B}$. Given the job's physical plan template, we learn $\lambda_{\mathbf{B}}^{\mathbf{A}}(\cdot)$ by simply observing the results of the lineage function $\Lambda_{\mathbf{B}}^{\mathbf{A}}(\mathcal{B})$ on different inputs. No execution of the job is needed.

Calculation of deficiencies follows a procedure similar to that of determining the set of deficient tiles in Section 3.3. We start with job j_{hit} and work backwards. Consider $\mathbf{A}$ produced by job j. If $\mathbf{A}$ is needed by any job following j_{hit}, $\chi^{\mathbf{A}} = 1 - \rho_{j_{\mathsf{hit}},\phi}^{\mathbf{A}}$. Otherwise, let $\mathcal{B}$ denote the set of matrices produced by any job among $j+1, j+2, \ldots, j_{\mathsf{hit}}$ that reads $\mathbf{A}$. For each $\mathbf{B} \in \mathcal{B}$, the probability that an $\mathbf{A}$ tile is needed in producing a $\chi^{\mathbf{B}}$ fraction of $\mathbf{B}$ tiles is $\lambda_{\mathbf{B}}^{\mathbf{A}}(\chi^{\mathbf{B}})$. Assuming independence, we have $\chi^{\mathbf{A}} = \big(1 - \prod_{\mathbf{B}\in\mathcal{B}}(1-\lambda_{\mathbf{B}}^{\mathbf{A}}(\chi^{\mathbf{B}}))\big)(1-\rho_{j_{\mathsf{hit}},\phi}^{\mathbf{A}})$.

With deficiencies calculated, we can now estimate the execution time of the recovery phase. We need to run a partial version of job j ($1 \le j \le j_{\mathsf{hit}}$) in the recovery phase if this job produces some matrix with non-zero deficiency. Let $\mathcal{O}_j$ denote the set of matrices produced by job j. Assuming independence and that recovering a given amount of deficiency requires the same fraction of the total work, we estimate the fraction of work in job j needed in the recovery phase to be $1 - \prod_{\mathbf{A}\in\mathcal{O}_j}(1-\chi^{\mathbf{A}})$. Therefore, we can estimate the total execution time of the recovery phase as

$$\widetilde{T_{\mathsf{rec}}} = \sum_{1\le j\le j_{\mathsf{hit}}} \mathfrak{T}(n_{\mathsf{prim}}, 0, q_j) \cdot \Big(1 - \prod_{\mathbf{A}\in\mathcal{O}_j}(1-\chi^{\mathbf{A}})\Big), \qquad (3)$$

where $\mathfrak{T}(n_{\mathsf{prim}}, 0, q_j)$ estimates the time it takes to run job j on the primary nodes only.

4.5 Putting It Together: Cost Estimation

All components are now in place for us to describe Cümülön's cost estimation procedure. Overall, our strategy is to first generate a "time table" for execution on a full-strength cluster assuming no hit. Then, we simulate multiple market price traces. For each trace, we determine the hit time, place it in the context of the full-strength execution time table, estimate the lengths of the recovery and wrap-up phases, and then the total cost. The costs obtained from the collection of simulated traces give us a distribution, allowing Cümülön to optimize expectation while bounding variance.

More precisely, we are given a baseline plan $Q = \{q_1, \ldots, q_m\}$ and n_{prim} primary nodes, a bidding strategy $(\hat{p}, n_{\mathsf{tran}})$ and the current market price p_0 of transient nodes, and a syncing strategy S.

1. We generate a full-strength execution time table $t_1 < t'_1 \le t_2 < t'_2 \le \cdots < t_m$, where t_j is the estimated time when we job j completes, and t'_j is the time when the optional *sync* associated with job j completes ($t_j = t'_j$ if $S(j) = \varnothing$). For convenience, let $t'_0 = 0$ and $t'_m = t_m$. We use the following recurrence:

$$\begin{cases} t_j = t'_{j-1} + \mathfrak{T}(n_{\mathsf{prim}}, n_{\mathsf{tran}}, q_j); \\ t'_j = t_j + \widetilde{T_{\mathit{sync}}^{S(j),j}} \text{ if } S(j) \ne \varnothing, \text{ or } t_j \text{ otherwise.} \end{cases}$$

2. Given p_0, we simulate a market price trace $p(t)$ up to $t = t_m$ using the market price model. If $\forall t \in [0, t_m) : p(t) \le \hat{p}$, we have "lucky" run without a hit, so $T_{\mathsf{hit}} = t_m$ and $T_{\mathsf{rec}} = T_{\mathsf{rap}} = 0$. Otherwise, we estimate T_{hit}, T_{rec}, and T_{rap} as follows:
   - $T_{\mathsf{hit}} = \min\{t \in [0, t_m) \mid p(t) > \hat{p}\}$.
   - $j_{\mathsf{hit}} = \max\{j \in [1, t_m] \mid t'_j < T_{\mathsf{hit}}\}$.
   - We estimate T_{rec} using Eq. (3). There are two cases: 1) If $T_{\mathsf{hit}} < t_{j_{\mathsf{hit}}}$, the cluster is in the middle of executing job j_{hit} when hit. We set ϕ for job j_{hit} to $(T_{\mathsf{hit}} - t'_{j_{\mathsf{hit}}-1})/(t_{j_{\mathsf{hit}}} - t'_{j_{\mathsf{hit}}-1})$. 2) Otherwise, job j_{hit} is completed but the cluster is in the middle of a *sync* job when hit. Thus, ϕ for job j_{hit} is 1, but ϕ for the following *sync* job is $(T_{\mathsf{hit}} - t_{j_{\mathsf{hit}}})/(t'_{j_{\mathsf{hit}}} - t_{j_{\mathsf{hit}}})$.
   - We estimate T_{rap} as $\sum_{j\in[j_{\mathsf{hit}}+1,m]} \mathfrak{T}(n_{\mathsf{prim}}, 0, q_j)$.

   Finally, using the estimated T_{hit}, T_{rec}, and T_{rap}, we calculate the cost using Eq. (1) for the given market price trace.

To account for uncertainty in the market price, we repeat Step 2 above multiple times to obtain a cost distribution. From this distribution, we can calculate the expected cost as well as the probability that the cost exceeds a certain threshold.

Currently, Cümülön does not account for uncertainty in the estimates of job execution times or data persistency/deficiency. For the linear algebra workloads targeted by Cümülön, we found such uncertainty to be manageable and dwarfed by the uncertainty in the market price. However, if available, more sophisticated models providing uncertainty measures can be incorporated into the procedure above in a straightforward way by sampling. For example, Step 1 can be repeated to obtain multiple samples of execution time tables. Doing so will increase the complexity of the cost estimation procedure by a multiplicative factor.

4.6 Putting It Together: Optimization

To choose a bidding strategy, Cümülön basically performs a grid search through all candidate pairs of $\hat{p}$ (bid price) and n_{tran} (number of transient nodes) values. The bid price starts at p_0 and is incremented by one cent at a time; the number of transient nodes starts from 0 and is incremented by one at time. We now discuss how to upper-bound these two parameters.

- Under the default pricing scheme in this paper, once $\hat{p}$ is high enough, additional increase in it will have very little impact—the distribution of T_{hit} will not improve much further, and the average market price paid for the transient nodes over time will not increase much further. Therefore, in our setting, after reaching $\hat{p} = 2p_{\mathsf{prim}}$ (twice the fixed price for the primary nodes), we

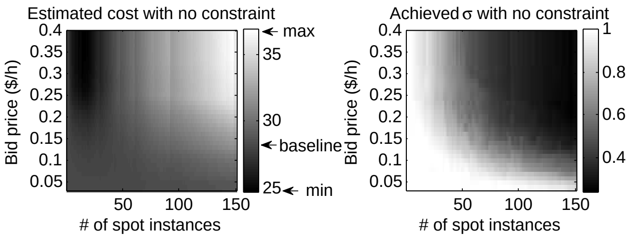


Figure 4: Optimal plans for MM_1^{50}, as we vary the bidding strategy. *No risk tolerance constraint* is set here. a) Intensity shows the estimated expected cost of the plan (darker is better). b) Intensity shows the probability that the cost is within 1.05 of the baseline (lighter is better).

stop increasing $\hat{p}$ if the resultant change in expected cost is less than 0.01%.

- A larger number of transient nodes leads to both diminishing returns of parallelization and a higher chance for the total cost to overrun the user-specified threshold. For workloads studied in this paper, we set the upper bound of n_{tran} to 150, which is enough to cover the optimal plans. For some workloads, a larger n_{tran} may well lower expected cost further, though we do not go beyond $n_{\text{tran}} = 300$ for practical reasons.

An obvious improvement to the search algorithm above would be to first search a coarser grid, and then search the more promising regions at the finer granularity. However, we found the simple algorithm to suffice for our workloads because the ranges of $\hat{p}$ and n_{tran} are limited in practice.

Given a physical plan template Q of m jobs and the cluster configuration $(n_{\text{prim}}, n_{\text{tran}})$, Cümülön uses the same procedure as Cumulon to choose the optimal number of tasks (and hence the amount of work per task) for each job. Next, Cümülön chooses the syncing strategy S. The search space of syncing strategy can be large as it is exponential in m. Another challenge is that adding a *sync* job does not always lower the expected total cost; nonetheless, even if a *sync* increases the expected cost, it may be useful as it reduces variance and decreases the tail probability of cost overrun. Cümülön resorts to a two-phase greedy strategy. In the first phase, we always pick the *sync* job (among the remaining options) that decreases the expected cost the most; we repeat this step until no more *sync* jobs can be added to S. If the risk tolerance constraint is met, we simply return S. Otherwise, we proceed to the second phase: we always pick the *sync* job that gives the biggest increase in the probability of cost staying within the prescribed threshold, and repeat this step until no more improvement can be made. The complexity of this greedy algorithm is only quadratic in m.

To illustrate Cümülön's optimization decisions, consider the following examples based on a synthetic program MM_γ^k. This program is simple by construction, so that we can control its characteristics and intuitively understand the trade-offs between various optimization decisions. Specifically, MM_γ^k computes $\mathbf{B} \times \mathbf{A}^k$ using a chain of k matrix multiplies, each multiplying the previous result by $\mathbf{A}$. By fixing the choice of splits (recall Section 2), we control the read factor γ of the intermediate result by the next multiply. Both $\mathbf{B}$ and $\mathbf{A}$ are 30720×30720 in the following examples.

Example 3 (MM_1^{50} Plan Space)**.** *Recall RSVD-1 in Example 1 and Figure 1. To see how different programs call for different plans, we now consider MM_1^{50} for comparison.*[3] *The baseline costs* $28.31 *and runs under* 65 *hours, using* 3 *machines of type* `c1.medium` *at the fixed price of* $0.145 *per hour. The current market price is* $0.02. *Again, we explore the plan space by varying the bidding strategy* $(\hat{p}, n_{\text{tran}})$*, but here we do not impose a risk tolerance constraint, so Cümülön simply looks for plans with the lowest expected cost. Figure 4a plots the expected plan cost, while Figure 4b plots the probability that the cost stays within* 1.05 *of the baseline.*

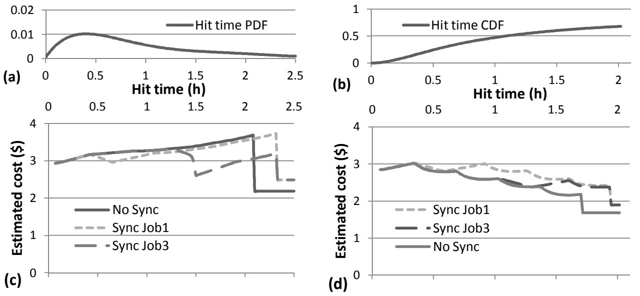


Figure 5: **Top:** Distribution of the hit time T_{hit}, with $p_0 = \$0.02$ and $\hat{p} = \$0.2$. a) PDF. b) CDF. **Bottom:** Expected total cost of MM_γ^5 as a function of T_{hit}. c) Low read factor ($\gamma = 1$). d) High read factor ($\gamma = 5$).

Figure 4 has a very different plan space compared with Figure 1. Instead of bidding for lots of transient nodes at a relatively low bid price for RSVD-1, here we want to bid for fewer transient nodes at a relatively high bid price. For example, bidding for 13 *nodes at* $0.37 *gives an expected cost of* $24.73*, with probability* 0.91 *of staying within* 1.05 *of the baseline. Intuitively, the jobs in MM_1^5 are less scalable. Therefore, larger clusters have diminishing effects on completion time, and this low cost-effectiveness drives up expected cost, as evidenced in Figure 4a. Also, larger clusters incur higher risks of cost overrun, as evidenced in Figure 4b.*

Example 4 (MM_γ^n Syncing Strategies)**.** *In this example, we examine how Cümülön chooses different syncing strategies based on a number of factors. First, we consider the distribution of T_{hit} predicted by our market price model, given $p_0 = \$0.02$ and $\hat{p} = \$0.2$. As explained earlier in Section 4.1, the top part of Figure 5 shows this distribution. Figure 5c plots the expected cost of MM_1^5 conditioned on T_{hit}, for three plans that differ only in their syncing strategies; $n_{prim} = 3$ and $n_{tran} = 10$. We make the following observations.*

- *For every syncing strategy, its plot always starts with the baseline cost ($T_{hit} = 0$); the last drop (before the plot become horizontal) always corresponds to a lucky run where the program finishes without being hit.*
- *Here, with a strategy of no syncing at all, we see that until the program finishes, there is a long "window of vulnerability" during which the cost is expected to raise steadily higher than the baseline, because a hit would take increasingly longer to recover as the intermediate result becomes more valuable over the course of execution.*
- *Adding a* sync *job increases the amount of work. Therefore, we see in Figure 5c that the cost of a lucky run is always higher than that of no* sync *at all. The benefit, however, is that a* sync *job can reduce the recovery cost, thereby lowering the expected total cost should a hit occur afterwards. Overall, they also tend to "smooth" the plot.*
- *Different* sync *jobs bring different benefits, and the choice matters. As Figure 5c shows, syncing after the first multiply helps very little with cost, because recovery is cheap initially anyway. Syncing before the last multiply (not shown here) would help the most, but the chance of realizing this gain is small because a hit would have to happen during the last job. Cümülön in this case*

[3] To get $\gamma = 1$ in MM_1^{50}, we conceptually partition each input into a grid of square submatrices, and let each task multiply a square submatrix of the intermediate result with the appropriate row of $\mathbf{A}$'s square submatrices. This choice of splits is in fact suboptimal, and the optimal setting is described later when $\gamma = 5$.

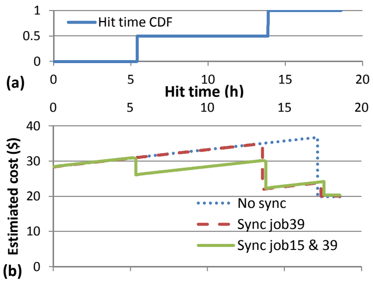


Figure 6: **Top:** CDF of a contrived T_{hit} distribution, with price peaking at two known time points. **Bottom:** Expected total cost of MM_1^{50} as a function of T_{hit}.

chooses to place sync *after the third multiply, which balances the two considerations.*

The reality is more complicated, as data caching during execution affects how much sync *jobs reduce recovery costs, and how much they cost themselves. To illustrate, consider instead MM_5^5, where we raise the read factor of the intermediate result.*[4] *Figure 5d again compares the three syncing strategies. Here, because of the higher read factor, even without any explicit* sync *jobs, most of the intermediate result becomes cached at the primary store during the following job, and will not need to recomputed during recovery. Thanks to this caching effect, as shown in Figure 5d, the strategy of no syncing performs well, with expected total cost generally below the baseline (except when execution just begins). In comparison, the two strategies with explicit* sync *jobs have higher expected total costs in this case, because the* sync *jobs have a much denser I/O pattern that can bottleneck the primary nodes. By modeling special I/O patterns (Section 4.3) as well as read factors and persistency (Section 4.4), Cümülön is able to choose the strategy of no syncing intelligently.*

Next, we turn to the effect of future market price on syncing strategies. Consider the longer program MM_1^{50}. Figure 6a shows the distribution of T_{hit} for a contrived market price model, which predicts price peaks at two time points during the program execution, each happening with probability 0.5*. Figure 6b compares the strategy of no syncing with those of syncing after the 38th multiply and of syncing after both the 15th and the 39th. Cümülön's greedy algorithm adds first the 39th, then the 15th, before returning the result syncing strategy as its decision. From Figure 6b, we see that the two chosen* sync *jobs are timed to occur immediately before the two possible hit time points, which makes intuitive sense.*

4.7 Summary and Extensions

In summary, choices of bidding and syncing strategies depend on many factors and require evaluating multiple trade-offs. The amount of information and level of knowledge required for intelligent decisions, as well as the complexity of the problem, make the automatic optimization a necessity. As discussed at the beginning of this section, we have made a number of assumptions in Cümülön to make the optimization manageable. We now discuss several extensions that overcome the limitations of these assumptions.

Delayed bidding means starting with only primary nodes and waiting until an opportune time (e.g., off-peak hours) to bid for transient nodes. This strategy is especially useful given a periodic market price model (Section 4.1). We have extended the Cümülön optimizer to find, given the current time, the amount of time to wait before placing the bid. For details and experimental evaluation of delayed bidding, see the technical report [7].

Sequential bidding allows Cümülön to bid for a new set of transient nodes after recovering from a hit. This strategy can be achieved simply by invoking the optimizer again when needed, with the remaining workload. However, in making that decision, our optimizer always assumes that it is placing the last bid. Further research is needed to assess how much this assumption negatively impacts the optimality of sequential bidding in practice; Section 5.3 presents one experiment where this impact seems limited.

Going beyond these extensions, we would like to support a collection of transient nodes of different machine types, acquired at different times, and with different bid prices. We would also like to act dynamically in response to the market, and exercise the option of releasing transient nodes voluntarily. While Cümülön can already handle a heterogeneous cluster in storage and execution, cost estimation and optimization techniques for more general settings are still under development and many open problems remain.

5 Experiments

We conduct our experiments on Amazon EC2. As mentioned in Section 3.1, we implement the dual-store design using two separate HDFS instances, with default replication factor set to 3. For brevity, let an $(n_{\text{prim}}, n_{\text{tran}})$ cluster denotes a cluster with n_{prim} primary nodes and n_{tran} transient nodes. Most workloads used in our experiments have been introduced earlier: **RSVD-1** (Example 1), **GNMF** (Example 2), and $\mathbf{MM}_{\gamma}^{k}$ (Section 4.6). In specifying matrix sizes, "k" denotes 1024.

5.1 Storage Design and I/O Policy

We compare our dual-store design and I/O policy (Section 3.1) with three other alternatives. The baseline of comparison is *write primary + no read cache*, which is the third strawman solution described in Section 3.1. The two other alternatives can be seen as Cümülön's dual-store design with different features removed: *write primary + read cache* caches data read from the other store, but always writes to the primary; *write home + no read cache* always writes to the home store, but does not cache data read from the other store.

Figure 7 shows the execution times under the four I/O policies for three workloads with varying characteristics: MM_5^1, which multiplies two 30k × 30k input matrices; ADD, which adds two 40k × 40k input matrices; and one iteration of GNMF with $(n, m, k) = (4780\text{k}, 250\text{k}, 100)$. We run all workloads on a $(3, 20)$ `c1.medium` cluster, with two slots per node (with one exception[5]). The transient store is initially empty in all settings. The *write primary + no read cache* baseline is always the worst performer, so we use its execution time to normalize the other alternatives' executions times for each workload.

From Figure 7, we see that Cümülön's I/O policy performs consistently the best; it makes effective use of caching to minimize the amount of traffic between the primary and transient nodes. Comparing the two alternatives between Cümülön and the baseline, we see that *write home + no read cache* performs much better than *write primary + read cache*. From a performance perspective, avoiding writes to the primary store is even more important than avoiding reads, because writes are more expensive (due to the replication factor) and more likely to cause bottlenecks. Furthermore, writing to the home store effectively distributes intermediate result data evenly across the entire cluster, making reading of such data more balanced and likely serviceable by a large transient store. This

[4] To get $\gamma = 5$, we conceptually partition each input into a 5×5 grid of square submatrices, and let each task multiply a pair of square submatrices.

[5] For MM_5^1, with two slots per primary node, *write primary + no read cache* failed to run because of congested I/O requests. Therefore, in this case we had to set zero slot per primary node, essentially dedicating the primary nodes as storage nodes.

observation justifies Cümülön's design decision of not aggressively pushing writes across stores, but instead relying on caching and judicious use of *sync* jobs.

5.2 Time Estimation

We now turn to the validation of Cümülön's time estimation methods. In this experiment, we run MM_5^5 on a $(3, 20)$ `c1.medium` cluster; there is a *sync* following job 3. We artificially generate a hit at the one of 11 time points during execution.[6] Then, we let the recovery and wrap-up phases take their courses and measure the actual total execution time including all phases. The chosen hit times test a wide range of scenarios, e.g., hitting jobs at different points of progress, hitting in the middle of a *sync*, hitting when recovery involves a lot of (or little) work, etc.

In Figure 8, we compare the measured execution times with the estimates produced by Cümülön, across different hit times. As hit time is pushed later, the total execution time generally decreases, because we get to finish more work with a full-strength cluster, leaving less work to the wrap-up phase (which executes only on the primary nodes). As we can see, Cümülön's execution time estimates are consistently accurate.

5.3 Optimization

For experiments in this section, the plans use `c1.medium` clusters, with two slots per node. The primary nodes have a fixed price of $\$0.145$ per hour, and we assume that the current market price of transient nodes is $\$0.02$ per hour, which is the most frequently price in our historical price data. For cost estimation (Section 4.5), we simulate $10{,}000$ market price traces using our price model (Section 4.1) trained from historical Amazon spot price data in the first six months' of 2014 for `c1.medium` in zone `us-east-1a`.

Effect of Number of Iterations in RSVD-1 In this experiment, we investigate how longer iterative workloads affect Cümülön's optimization decisions. We consider RSVD-1 with $l = 2\text{k}$, $m = n = 100\text{k}$, and vary the number of iterations k from 1 to 15. Figure 9 shows, for each k, the cost of the baseline plan P_{base}, which uses 3 primary nodes, as well as the expectation and distribution of the cost of the optimal plan P_{opt} using transient nodes—subject to the risk tolerance constraint of $\delta = 0.05$ and $\sigma = 0.9$ (i.e., with probability no less than 0.9 the cost is no more than 1.05 times the baseline). The distribution is shown as vertical stripe of PDF where darker shades indicate higher densities. Additional details about P_{opt} are shown in Table 1.

An interesting observation is that the cost distribution of each P_{opt} appears roughly bimodal. The two density concentrations correspond to two possibilities: either we experience a hit or not during execution. If we are lucky to finish the workflow without a hit, we end up with a much lower cost than the baseline, because most work is done with cheaper transient nodes. However, if we are unlucky, we may incur extra recovery cost. Depending on how much we get done in the transient nodes, the overall cost might be higher or lower than the baseline. Nonetheless, because Cümülön observes the risk tolerance constraint in finding P_{opt}, it is very unlikely that we end up paying much higher than the baseline.

As the number of iterations increases, we see that the baseline cost increases proportionally as expected. For the cost distribution of P_{opt}, we see density gradually shifting from the lucky (lower-cost) to the unlucky (higher-cost) region, because we are more likely to encounter a hit before finish. Furthermore, from Table 1, we see that as the workflow becomes longer, we tend to bid for more transient nodes at lower prices, up to a certain a point when the bidding strategy stabilizes; meanwhile, the syncing strategy gradually injects more *sync* jobs and at later times during the workflow, which helps limit the recovery cost.

# iters	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\hat{p}$	0.30	0.21	0.06	0.18	0.16	0.13	0.13	0.10	0.11	0.10	0.09	0.09	0.10	0.10	0.09
n_{tran}	39	58	72	81	97	81	81	90	80	84	88	90	75	75	81
sync jobs	∅	∅	∅	∅	∅	1	1	2	2	2	3	3	3	3,8	4,9

Table 1: Bidding and syncing strategies chosen by the optimal plans of RSVD-1 for varying number of iterations.

It is worth noting the expected amount of cost saving (i.e., the gap between dotted and solid lines in Figure 9 converges to around $\$1.50$ as the number of iterations reaches 5. However, keep in mind that P_{opt} is limited by one bid only. There is a possibility of achieving more savings if Cümülön is allowed to bid again after a hit and when the market price comes down (Section 4.7 discusses such extensions). We still need further study of whether the one-bid optimization assumption is appropriate in this setting, but interestingly, in this particular case, the strategy obtained for 5 iterations (when cost saving converges) turns out to be not so different from those for more iterations, so the assumption would work well in this case.

Effect of Bid Price on GNMF Using GNMF, We now examine how the choice of bid price influences cost. Recall from Example 2 that each GNMF iteration is compiled into 6 Cümülön jobs. In this experiment, we consider two GNMF iterations running on a $(3, 10)$ cluster, with three different bid prices: $\$0.05$, $\$0.12$, and $\$0.25$ per hour. Cümülön picks the best plan with the given bidding strategy.[7] Figure 10 plots, for each of the three bid prices, the estimated cost distribution as well as expected cost over different hit times. The density reflects both the probability of the hit occurring at a given time and the probability of incurring a given amount of cost conditioned on the hit time.

Note that for all three cases in Figure 10, the average cost curve starts with the baseline cost and has four bumps. It turns out that the first and third drops correspond to *sync* jobs Cümülön places[8] after jobs 3 and 9 (recall Figure 3) to persist $\mathbf{H}'$ for the next iteration and for output, respectively. The second drop corresponds to the earlier waves of job 7, which effectively persists much of $\mathbf{W}'$ because of the high read factor. The last drop corresponds to the lucky case where no hit occurs during execution.

If we bid low at $\$0.05$, the hit will likely occur soon and we end up with a cost slightly higher than the baseline; with low probability, the transient nodes could survive longer and the cost would go down dramatically. If we bid higher, at $\$0.15$ or $\$0.25$, we are more likely to hold the transient nodes longer, hence the shift of density to later hit times. In particular, the probability of getting lucky (no hit during execution) becomes higher, as evidenced by the high-density regions around the end of average cost curve. On the other hand, with higher bid prices, the average market price we expect to pay over time also increases. As a result, the average cost curve no longer drops as dramatically as the case of bidding at $\$0.05$. In other words, since the spot instances are in expectation more expensive when we bid higher, they might not necessarily reduce cost even if we can keep them for longer duration.

Note on Optimization Time Even though Cümülön derives the full cost distribution of each possible plan by repeated simulation of the market price model, the total optimization time is reasonable for workloads whose sizes warrant the use of clouds for parallel

[6] Note that artificial hit injection gives us control over T_{hit}, allowing us to target different scenarios easily. If we run against the real market prices, getting the same level of test coverage would be far more expensive. Since our goal here is to validate time estimation, price is irrelevant.

[7] If we let Cümülön pick the bid price in this case, the optimal bid price will be $\$0.19$, which achieves an expected cost of $\$4.37$, compared with the baseline cost of $\$4.75$.

[8] The reason why Cümülön makes these choices is quite subtle; for more discussion, see the technical report [7].

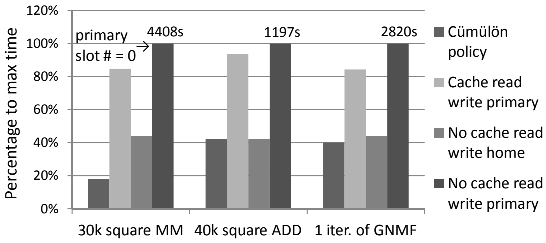


Figure 7: Comparison of alternative I/O policies for the dual-store design, across three workloads.

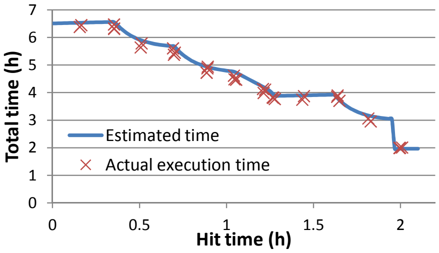


Figure 8: Estimated vs. actual total execution time of MM_5^5 as a hit occurs at different times.

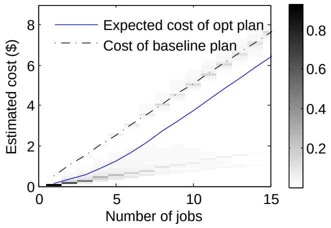


Figure 9: Estimated costs of optimal plans for RSVD-1 with varying number of jobs, vs. costs of the baseline plan

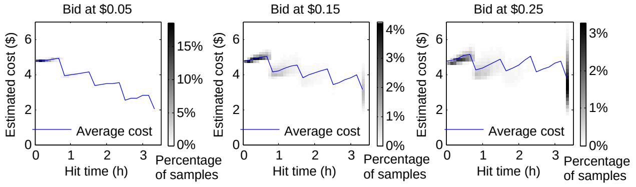
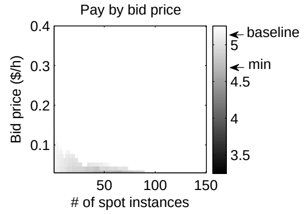


Figure 10: Effect of bid price on the distribution of hit time and total cost for two GNMF iterations.



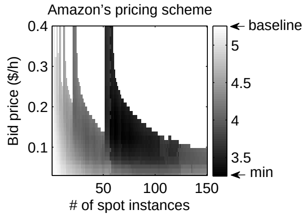


Figure 11: Estimated expected cost of the optimal plans for RSVD-1 given various bidding strategies, under the pay-by-bid-price scheme.

Figure 12: Estimated expected cost of the optimal plans for RSVD-1 given various bidding strategies, under Amazon's pricing scheme.

execution. For example, Figures 1, 4, 11, and 12 all require full optimization including the choices of bidding and syncing strategies; in every case, our optimizer completes under 6 minutes on a standard desktop computer with 4-core 3.4GHz Intel i7-2600 CPU and 8GB of memory.

5.4 Alternative Pricing Schemes

Paying by Bid Price What happens if we change our pricing scheme to charge transient nodes at their bid prices instead of the varying market price? Thanks to Cümülön's flexibility, we can simply plug in this pricing scheme and see how that it would affect our optimal strategies. Consider the same RSVD-1 workload of Example 1 and recall the plan space shown in Figure 1a. We keep all settings the same but change the pricing scheme; the resulting plan space is shown in Figure 11. As we can see, paying by bid price makes transient nodes significantly less attractive. Bidding high now has the disadvantage of directly increasing the cost of transient nodes (as opposed to increasing the probability of paying more under a pay-by-market-price scheme). Bidding above the fixed price of primary nodes no longer makes sense. The space of plans satisfying the risk tolerance constraint is much smaller compared with Figure 1a. The optimal plan is to bid low for 62 transient nodes at $\$0.03$ ($p_0 = \$0.02$), which achieves an expected cost of $\$4.66$ compared to the baseline of $\$5.09$. From the perspective of a cloud provider, if pay-by-bid-price scheme were to be adopted, the transient nodes would need to be priced much lower. For example, consistently halving the market price would make transient nodes comparably attractive as they were under the pay-by-market-price scheme (see [7] for more details).

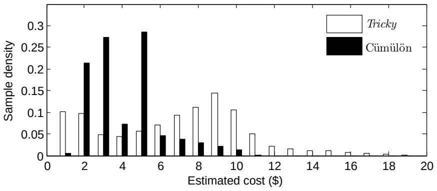


Figure 13: Comparison of the estimated cost distributions for Cümülön's optimal plan and *tricky* for RSVD-1.

Amazon's Pricing Scheme As discussed in Section 4.2, Amazon EC2 uses a pricing scheme different from what we assume by default in this paper. Amazon's scheme rounds usage time to full hours; for spot instances, it does not charge for the last partial hour of usage if they are hit. Under this pricing scheme, we let Cümülön explore the plan space for the same RSVD-1 workload considered in Figures 1a and 11 under the same settings. The result is shown in Figure 12. The feasible plan space now has a jagged boundary because of usage time rounding: the spikes corresponds to cases when an increase in cluster size allows the workflow to complete just before the last hour ends. Also, thanks to the free last partial hour when hit, the optimal plan in this case—which bids high ($0.15) and big (for 60 transient nodes) and syncs two jobs—can achieve a lower expected cost ($\$3.25$) than with the default pricing scheme ($\$5.22$).

Since Cümülön's optimal plan in this case is far from intuitive, it is interesting and instructive to compare this plan with what a "tricky" user might do. Intuitively: **1)** Let us bid low—exactly at the market price—so either we get the work done at a very low cost, or a hit happens and the last partial hour is free anyway. **2)** We will bid again after a hit (Cümülön's plan only bids once), as soon as the market price is lower than the fixed price of the primary nodes. **3)** We will play a (not-so-nice) trick: even after the workflow has completed, we will keep the cluster until the end of the current hour, because if a hit happens we will get that hour for free (even if a hit does not happen, we will not pay more because of rounding). **4)** Because of the higher hit probability, we sync after every job. **5)** We use the same cluster size as Cümülön, i.e., we have 3 primary nodes and always bid for 60 transient nodes.

We compare the cost distributions of Cümülön's plan and this "tricky" strategy (thereafter referred to as *tricky*) in Figure 13. Overall, *tricky* has an expected cost of $\$6.86$, much higher than Cümülön's $\$3.25$, and in fact higher than the baseline of $\$5.22$. Furthermore, *tricky* exhibits larger variance. Thanks to the first three of its features above, *tricky* does have a higher probability than Cümülön of achieving very low costs. On the other hand, *tricky* incurs considerable costs in syncing after every job and in getting data out of the primary store after acquiring new transient nodes, both of which bottleneck the primary nodes. The advantages of bidding low are offset by the increased recovery cost, and there is only small chance of getting the last hour for free holding the cluster after completion.

This comparison highlights the difficulty in manually devising bidding strategies and illustrates the effectiveness of Cümülön optimization despite its various assumptions.

6 Related Work

Previous work dealt with the unreliability of transient nodes in two ways. The first is to use a storage system capable of handling massive correlated node failures. For example, *Glacier* [4] uses high degrees of redundancy to achieve reliability; *Spot Cloud MapReduce* [11] depends on a reliable external storage service rather than local storage in the cluster. Both methods have negative performance implications. Like Cümülön, a number of systems use more reliable primary nodes for storage. Chohan et al. [3] deploys a HDFS only on the primary nodes, and uses transient nodes for computation only; Amazon's Elastic MapReduce clusters can also be configured in this fashion. This method has to limit the number of transient nodes, because the primary nodes can easily become an I/O bottleneck when outnumbered. Going a step further, *Qubole*'s auto-scaling cluster deploys the storage system on all nodes, but with a customized data placement policy to ensure that at least one replica is stored among primary nodes; *Rabbit* [1] is a multi-layer storage system that can be configured so that one replica goes to the primary nodes. However, as we have discussed (Section 3.1) and verified (Section 5.1), writing all data to the primary nodes still causes unnecessary performance degradation.

The second way of dealing with unreliable transient nodes is to checkpoint them. A lot of previous work [16, 17, 19, 2, 15, 8, 20] studied checkpointing and bidding strategies under various settings in order to satisfy service level agreements, meet deadlines, or minimize cost. Others [12, 14] considered how to maximize the profit of a service broker who rent transient nodes and run workloads for users. All work above relied on external storage service for checkpointing. Their execution time models were rather simplistic—jobs have given, fixed execution times and are amenable to perfect scaling (if parallelization is considered). Moreover, they targeted general workloads and were therefore limited in their options—essentially, they must checkpoint the entire execution state and recover from the last completed checkpoint. With additional knowledge about the workload and lineage tracking, systems such as *Spark* [21] are able to infer which units of computation to rerun in order to recover from failures. As discussed in Section 3, thanks to declarative program specification, Cümülön has more intelligent checkpointing and recovery: its syncing strategy is selective, driven by a cost/benefit analysis informed by the market price model; its recovery is more flexible and precise in avoiding unnecessary computation, and does not require tracking lineage explicitly.

While Cümülön aims at helping users of a public cloud, others [23, 18] have approached the issue of spot instances from the cloud provider's perspective, seeking to maximize its profit by dividing its resource into different types (e.g., on-demand vs. spot) and pricing them optimally. Our work is complementary; cloud providers can also gain insights from our what-if analysis of pricing schemes (Section 5.4).

7 Conclusion

In this paper we have presented Cümülön, a system aimed at helping users develop and deploy matrix-based data analysis programs in a public cloud, featuring end-to-end support for spot instances that users bid for and pay for at fluctuating market prices. Cümülön's elastic computation and storage engine for matrices makes effective use of highly unreliable spot instances. With automatic cost-based, risk-aware optimization of execution, deployment, bidding, and syncing strategies, Cümülön tackles the challenge of how to achieve lower expected cost using cheap spot instances, while simultaneously bounding the risk due to uncertainty in market prices.

While Cümülön focuses on matrix computation, many of our techniques carry over to other data-intensive workloads expressed in high-level, declarative languages. For black-box workloads, techniques such as the dual-storage design and the overall risk-aware optimization algorithm still apply, but cost estimation becomes considerably more difficult and the errors and uncertainty therein must be accounted for together with the uncertainty in market prices. Generalization of the Cümülön approach will be an interesting direction to further explore.

References

[1] H. Amur, J. Cipar, V. Gupta, G. R. Ganger, M. A. Kozuch, and K. Schwan. Robust and flexible power-proportional storage. In *2010 ACM Symposium on Cloud Computing*.

[2] A. Andrzejak, D. Kondo, and S. Yi. Decision model for cloud computing under sla constraints. In *2010 IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*.

[3] N. Chohan, C. Castillo, M. Spreitzer, M. Steinder, A. Tantawi, and C. Krintz. See spot run: using spot instances for mapreduce workflows. In *2010 HotCloud*.

[4] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *2005 USENIX Symposium on Networked Systems Design and Implementation*.

[5] B. Huang, S. Babu, and J. Yang. Cumulon: Optimizing statistical data analysis in the cloud. In *2013 ACM SIGMOD International Conference on Management of Data*.

[6] B. Huang, N. W. Jarrett, S. Babu, S. Mukherjee, and J. Yang. Cumulon: Cloud-based statistical analysis from users perspective. *2014 IEEE Data Engineering Bulletin*.

[7] B. Huang, N. W. Jarrett, S. Babu, S. Mukherjee, and J. Yang. Cümülön: Matrix-based data analytics in the cloud with spot instances. Technical report, Duke University, Mar. 2015. `http://db.cs.duke.edu/papers/HuangJarrettEtAl-15-cumulon-spot.pdf`.

[8] H. Huang, L. Wang, B. C. Tak, L. Wang, and C. Tang. Cap3: A cloud auto-provisioning framework for parallel processing using on-demand and spot instances. In *2013 IEEE Cloud*.

[9] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *2007 EuroSys Conference*.

[10] D. D. Lee and H. S. Seung. Algorithms for non-negative matrix factorization. In *2000 Advances in Neural Information Processing Systems*.

[11] H. Liu. Cutting mapreduce cost with spot market. In *2011 HotCloud*.

[12] M. Mazzucco and M. Dumas. Achieving performance and availability guarantees with spot instances. In *2011 IEEE 13th International Conference on High Performance Computing and Communications*.

[13] V. Rokhlin, A. Szlam, and M. Tygert. A randomized algorithm for principal component analysis. *2009 SIAM Journal on Matrix Analysis and Applications*.

[14] Y. Song, M. Zafer, and K.-W. Lee. Optimal bidding in spot instance market. In *2012 IEEE International Conference on Computer Communications*.

[15] M. Taifi, J. Y. Shi, and A. Khreishah. Spotmpi: A framework for auction-based hpc computing using amazon spot instances. In *2011 Algorithms and Architectures for Parallel Processing*.

[16] S. Tang, J. Yuan, and X.-Y. Li. Towards optimal bidding strategy for amazon ec2 cloud spot instance. In *2012 IEEE Cloud*.

[17] W. Voorsluys and R. Buyya. Reliable provisioning of spot instances for compute-intensive applications. In *2012 IEEE 26th International Conference on Advanced Information Networking and Applications (AINA)*.

[18] P. Wang, Y. Qi, D. Hui, L. Rao, and X. Liu. Present or future: Optimal pricing for spot instances. In *2013 IEEE 33rd International Conference on Distributed Computing Systems (ICDCS)*.

[19] S. Yi, A. Andrzejak, and D. Kondo. Monetary cost-aware checkpointing and migration on amazon cloud spot instances. *2012 IEEE Transactions on Services Computing*.

[20] M. Zafer, Y. Song, and K.-W. Lee. Optimal bids for spot vms in a cloud for deadline constrained jobs. In *2012 IEEE Cloud*.

[21] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *2012 USENIX Symposium on Networked Systems Design and Implementation*.

[22] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *2010 HotCloud*.

[23] Q. Zhang, Q. Zhu, and R. Boutaba. Dynamic resource allocation for spot markets in cloud computing environments. In *2011 Fourth IEEE International Conference on Utility and Cloud Computing (UCC)*.