Cumulon: Simplified Matrix-Based Data Analytics in the Cloud

by

Botong Huang

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Jun Yang, Co Supervisor

_____
Shivnath Babu, Co Supervisor

_____
Sayan Mukherjee

_____
Alvin R. Lebeck

Dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the Department of Computer Science
in the Graduate School of Duke University
2016

Abstract

Cumulon: Simplified Matrix-Based Data Analytics in the Cloud

by

Botong Huang

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Jun Yang, Co Supervisor

_____
Shivnath Babu, Co Supervisor

_____
Sayan Mukherjee

_____
Alvin R. Lebeck

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University
2016

Abstract

Cumulon is a system aimed at simplifying the development and deployment of statistical analysis of big data in 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 cloud software platforms. Given user-specified requirements in terms of time, monetary cost, and risk tolerance, Cumulon automatically makes intelligent decisions on 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 them. Cumulon explores two alternative approaches toward supporting such markets, with different trade-offs between system and optimization complexity. Experimental study is conducted to show the efficiency of Cumulon's execution engine, as well as the optimizer's effectiveness in finding the optimal plan in the vast plan space.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank my colleagues, friends and families for their great support for my PhD study. I cannot get here without them.

I want to give special thanks to my primary advisor Professor Jun Yang, who introduced me to cutting-edge research, and provided enormous and continuous guidance and advice to me at all times throughout my PhD career. He has always been my role model.

I also want to thank my secondary advisor Professor Shivnath Babu for his supervision and contribution in my research career.

Thank Nicholas W.D. Jarrett for his great help developing the price model for Cümülön (in Appendix B). It is a good experience collaborating with him.

I am grateful to my committee members Professor Sayan Mukherjee and Alvin Lebeck, and also Professor Ashwin Machanavajjhala and Sudeepa Roy for their comments and suggestions to my thesis work.

Furthermore, I would like to take this opportunity to thank Microsoft Azure cloud team, Google Ads backend team and the SystemML group in IBM Research Alamden for hosting me for a summer internship. I have gained both technical and industrial experience out of them.

Lastly, I would like to thank the NSF, Duke Graduate School, and Duke department of Computer Science for funding my research during the PhD study.

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 according to clear price tags (e.g., Table 1.1), without worrying about acquiring, hosting, and maintaining physical hardware. Users also benefit from simpler programming models (the most prominent example being MapReduce [12], which Hadoop implements) and a wealth of tools, some of which have started to target machine learning and statistical computing—Apache Mahout, Greenplum/MADlib [10, 17], HAMA [36], RHIPE [15], Ricardo [11], Haloop [7], SystemML [14], ScalOps [5], Tensorflow [1] just to name a few.

Table 1.1: A sample of machine types available in Amazon EC2 and their on-demand prices.

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

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, 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. The emerging of aforementioned libraries and platforms for statistical analysis and machine learning helps 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. Now yet another question is: should we get the on-demand instances, bid for spot instances, or do both? Overall, 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 end-to-end solution aimed at simplifying the development and deployment of matrix-based data analysis in the cloud. When developing such programs, users of Cumulon will be able to think and code in a natural way using the familiar language of linear algebra. A lot of statistical data analysis (or computationally expensive components thereof) can be succinctly written in the matrix notation and with basic operations such as multiply, add, transpose, etc. Here are two examples that we will revisit later:

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

Data: $\mathbf{O}$: $m \times n$ sparse word-document matrix, where m is the vocabulary size and n is the number of documents;
k: number of latent topics (much less than m and n).
Result: $\mathbf{B}$: $n \times k$ dense document-topic matrix;
$\mathbf{C}$: $m \times k$ dense word-topic matrix;
$\mathbf{E}$: $k \times k$ diagonal matrix representing topic frequencies.

initialize $\mathbf{B}, \mathbf{C}, \mathbf{E}$;
repeat
  $\mathbf{F} \leftarrow \mathbf{O} \circ\!\!/ (\mathbf{C} \times \mathbf{E} \times \mathbf{B}^\intercal)$;
  $\mathbf{E}' \leftarrow \mathbf{E} \circ (\mathbf{C}^\intercal \times \mathbf{F} \times \mathbf{B})$;
  $\mathbf{C}' \leftarrow (\mathbf{C} \times (\mathbf{I} \circ\!\!/ \mathbf{E}')) \circ (\mathbf{F} \times \mathbf{B} \times \mathbf{E})$;
  $\mathbf{B}' \leftarrow (\mathbf{B} \times (\mathbf{I} \circ\!\!/ \mathbf{E}')) \circ (\mathbf{F}^\intercal \times \mathbf{C} \times \mathbf{E})$;
  $\mathbf{B}, \mathbf{C}, \mathbf{E} \leftarrow \mathbf{B}', \mathbf{C}', \mathbf{E}'$;
until *termination condition*;

- Singular value decomposition (SVD) of a matrix has extensive applications in statistical analysis. In the randomized algorithm of [35] for computing an approximate SVD, the first (and most expensive) step involves a series of matrix multiplies. Specifically, given an $m \times n$ input matrix $\mathbf{A}$, this step uses an $l \times m$ randomly generated matrix $\mathbf{G}$ whose entries are i.i.d. Gaussian random variables of zero mean and unit variance, and computes $\mathbf{G} \times (\mathbf{A} \times \mathbf{A}^\intercal)^k \times \mathbf{A}$. Typically, l is much smaller than m and n, and k is small (say 5). We refer to this step as RSVD-1.

- A popular method in information retrieval and text mining is *Probabilistic Latent Semantic Indexing* (*PLSI*) [20]. The algorithm can be conveniently written in the matrix notation with just a few lines, as shown in Algorithm 1.

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

When deploying such programs, users should be able to specify their objectives and constraints in straightforward terms—time, money, and risk tolerance. Then,

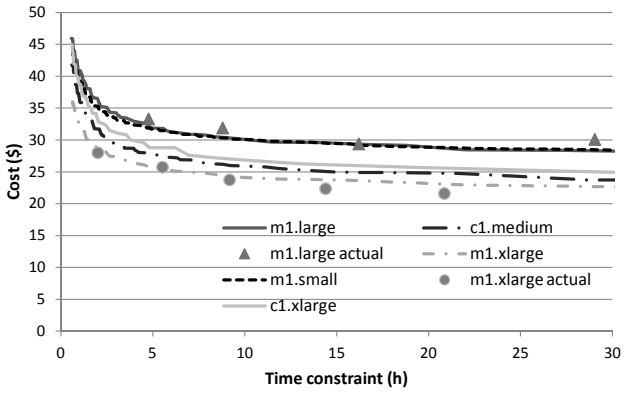

FIGURE 1.1: Costs of the optimal deployment plans for RSVD-1 (with l = 2,000, $m = n$ = 204,800, $k = 5$) using different Amazon EC2 on-demand machine types under different time constraints. Curves are predicted; costs of actual runs for sample data points are shown for comparison.

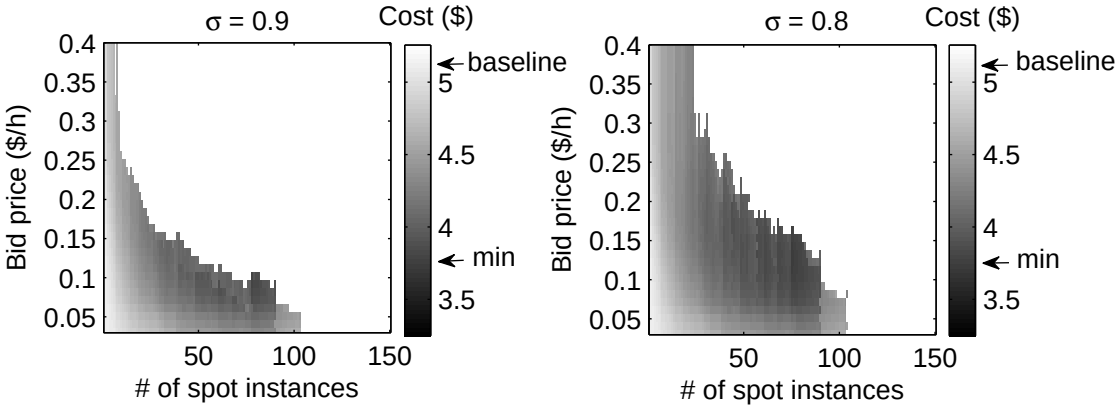

FIGURE 1.2: Estimated expected cost of the optimal plans for RSVD-1 (with l = 2,000, $m = n$ = 102,400, $k = 5$), as we vary the bid price and the number of spot instances we bid for. The cost is shown using intensity, with darker shades indicating lower costs. In the upper-right region, plans fail to meet the user-specified risk tolerance; hence, this region shows the baseline cost of the optimal plan with no bidding. All machines are of type `c1.medium`.

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 1.1 shows the costs[1] of best plans for RSVD-1 as we vary the constraint on expected completion time, if we only consider using the on-demand instances. 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.

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 *reclaimed* (taken away).[2] The user cannot change the bid price once the bid is placed. For example, consider again RSVD-1.[3] It takes a baseline cost of $5.09 to run under 11.7 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

[1] To simplify the interpretation of results here, we assume users are charged by fractional hours, rather than following Amazon EC2's practice of rounding usage time to full hours (though it is straightforward for Cumulon to use that pricing policy).

[2] Amazon EC2 actually does not charge for partial hour of usage of spot instances if they are reclaimed (as opposed to round usage time up to full hours if terminated voluntarily by users). This policy is specific to Amazon and can lead to some rather interesting results. To avoid making our results too specific to Amazon, we consider fractional hours by default when computing costs throughout the thesis and make no special case for spot instances. (Again, Cumulon can readily support the actual Amazon policy and results will be shown in the experiments.)

[3] Please note that there are minor differences between the workload settings and prices in the examples and figures of the thesis, because the results were originally reported in different papers [21, 25] and obtained at times when Amazon charged different prices.

baseline. Because of the uncertainty in future market prices, Cumulon lets users specify a risk tolerance—e.g., *"with probability no less than $\sigma = 0.9$, the overall cost will not exceed the baseline by more than $\delta = 5\%$."* Figure 1.2 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 \$0.10 per hour each (versus the current market price of \$0.02), which would reduce the expected overall cost to \$3.78 while staying within the user's risk tolerance.

1.1 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* [12], 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 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.

1.2 Road map

The rest of the chapters are organized as follows. Chapter 2 introduces Cumulon's efficient and flexible execution model, and demonstrates its benefits over "traditional" Hadoop-based solutions. We will then talk about the cost models in Cumulon optimizer and how it produces the optimal plan that only considers on-demand nodes, given a user specified constraint, e.g. minimize monetary cost given a deadline.

Next, we will discuss Cumulon's support for auction-based computing resources, i.e., spot instances. We refer to the Cumulon versions with spot instance support as **Cümülön**. Because of the potential problem of sudden correlated massive reclamation of the transient nodes, the storage system need some persistant storage to (partially) preserve data and states, in order to ensure progress. In general, we can

either rely on on-demand nodes or some external storage system, for instance *Amazon S3* or an HDFS running on some other reliable nodes. We tried both approaches in Cümülön v1 and Cümülön v2.

In Cümülön v1, the system consists of a primary cluster of reliable on-demand nodes and a transient cluster of spot instances. A dual-store storage system is deployed on top of both clusters. Execution starts with a "baseline" plan that only uses fixed-price on-demand instances. This is the optimal plan generated by Cumulon that meets the user-specified completion deadline and minimizes expected monetary cost. Next, Cümülön v1 makes intelligent decisions on how to bid for additional spot instances, and how to use them effectively to reduce expected cost while staying within users' risk tolerance. We will discuss Cümülön v1 in Chapter 3.

Cümülön v1 comes with a smart and efficient system design, but the system complexity makes the performance harder to predict; consequently, we limit Cümülön v1's optimizer to consider one single batch of spot instances at any time. Now we ask: is there a better trade-off between system complexity and optimizability? How about a simpler and easier-to-model system that allows a smarter optimizer to support dymanic bidding and multiple batches of spot instances?

With this in mind, we took a different approach in Cümülön v2. Assuming there is an external scalable storage system that can store all data, we no longer need to worry about data loss any more. Cümülön v2 simply maintains a dynamic pool of spot instances for execution. It dynamically adjusts the number of machines according to the observed market price, in order to minimize the total execution cost under acceptable cost variance, given a user-specified deadline. The optimization problem is formulated as a Markov Decision Process (MDP) in finite time space. The solution of the MDP model is a policy that instructs the optimal action (cluster size) to take in every possible future state. Later in Chapter 4 we will elaborate on Cümülön v2's design.

2

Cumulon

2.1 System Overview and Road Map

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

Logical Plan and Rewrites Cumulon first converts the program into a *logical plan*, which consists of *logical operators* corresponding to standard matrix operations, e.g., matrix transpose, add, multiply, element-wise multiply, power, etc. Then, Cumulon conducts a series of rule-based *logical plan rewrites* to obtain "generally better" logical plans. Such rewrites consider input data characteristics (e.g., matrix sizes and sparsity) and apply linear algebra equivalences to reduce computation or I/O, or increase parallelism.

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

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

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

Physical Operators and Plan Templates For each logical plan, Cumulon translates it into one or more *physical plan templates*, each representing a parallel execution strategy for the program. A physical plan template represents a workflow of *jobs*, where each job is in turn a DAG (directed acyclic graph) of *physical operators*. We will provide more details on these concepts in Section 2.2. For now, think of a job as a unit of work that is executed in a data-parallel fashion, where each thread of execution, or *task*, runs an instance of the physical operator DAG in a pipelined fashion over a different input split. The translation procedure attempts to group operators into as few jobs as possible, but if doing so causes one job to contain multiple resource-hungry physical operators, alternatives will be generated. Also note that one logical operator may be translated into one or more physical operators, and sometimes one physical operator may implement multiple logical operators.[1]

For example, Figure 2.1 shows two possible physical plan templates for $\mathbf{A}\times\mathbf{B}+\mathbf{C}$. The first one (a) has one job while the second one (b) has two. The reason why two jobs might be preferable will become apparent when we discuss in Section 2.2 how Cumulon implements matrix multiply in detail. Roughly speaking, the physical operator *Mul* only multiples submatrices from $\mathbf{A}$ and $\mathbf{B}$. In general, to complete $\mathbf{A}\times\mathbf{B}$, these results need to be further grouped and aggregated; in Figure 2.1b, *Add* performs this aggregation (as well as addition with $\mathbf{C}$) in a separate job. On the other hand, Figure 2.1a applies in the special case where submatrices multiplied

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

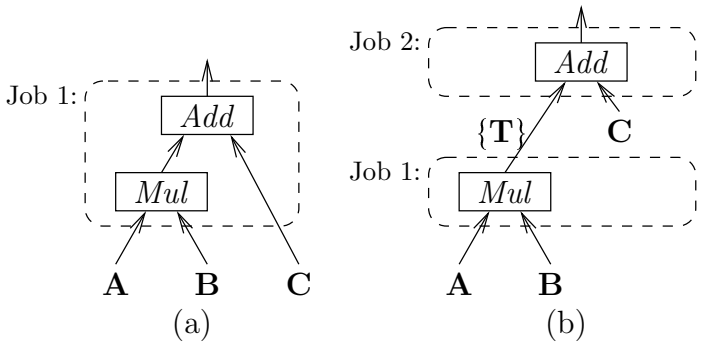

FIGURE 2.1: Two physical plan templates for $\mathbf{A} \times \mathbf{B} + \mathbf{C}$. In (b), the output from *Mul*, $\{\mathbf{T}\}$, is a list of matrices whose sum is $\mathbf{A} \times \mathbf{B}$.

by *Mul* include complete rows from $\mathbf{A}$ and complete columns from $\mathbf{B}$; no further aggregation is needed and *Add* can perform addition with $\mathbf{C}$ in the same job as *Mul*.

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

- $\mathcal{L}$ is a physical plan template.
- $\mathcal{O}$ represents parameters settings for all physical operators in $\mathcal{L}$. For example, for the *Mul* operator in Figure 2.1b, we need to specify the sizes of the submatrices multiplied in each task.
- $\mathcal{P}$ represents hardware provisioning settings, e.g., what machine type and how many machines to reserve on Amazon EC2. In general, such settings may change during the course of execution; for example, when a program shifts from an I/O-intensive stage to a computation-intensive stage, we may want to switch to fewer machines with more powerful CPUs.
- $\mathcal{Q}$ represents configuration settings. For example, for a Hadoop-based implementation of Cumulon, we need to specify the number of map/reduce "slots"

per machine. In general, these setting can also change during the course of execution.

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

Road Map Section 2.2 presents Cumulon's storage and execution models, and illustrates how to support matrix-based programs with physical plan templates and operators. We show the advantage of our model over MapReduce, and how to implement it conveniently on top of Hadoop despite its departure from MapReduce.

Section 2.3 discusses Cumulon's cost-based optimization for finding the best deployment plan given $\mathcal{L}$. We show how to estimate the costs of deployment plans in the cloud, and how to search the vast space of possible deployment plans efficiently.

Section 2.4 briefly discusses two other practical issues that arise in supporting statistical data analysis in the cloud: how to handle iterative programs, and how to cope with performance variances.

We then present experimental results in Section 2.5, discuss related work in Section 2.6, and conclude in Section 2.7.

2.2 Storage and Execution

As MapReduce has been the prevalent model for data-parallel programming in the cloud, a natural question is whether we can build our support for matrix-based data analysis on this model. Before presenting our approach for Cumulon in Section 2.2.2,

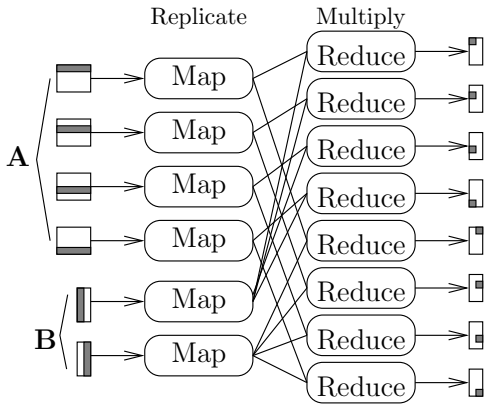

FIGURE 2.2: MapReduce job for $\mathbf{A} \times \mathbf{B}$ using SystemML's RMM; $f_l = 1$ is required.

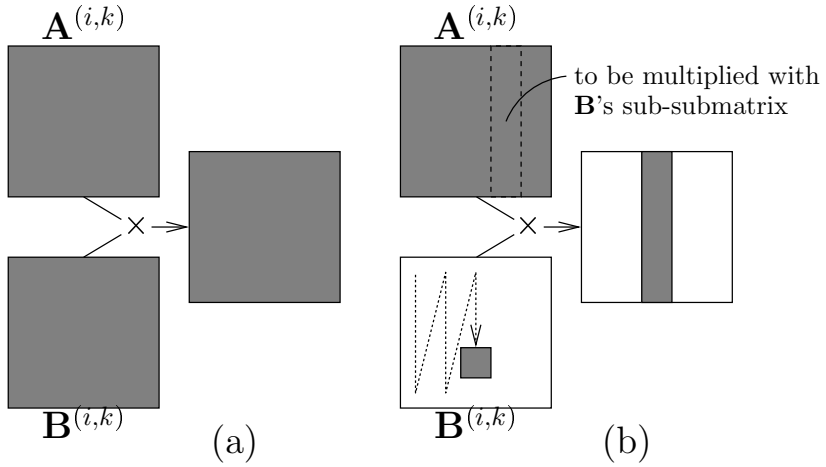

FIGURE 2.3: Options for multiplying submatrices within a task. Shaded portions are kept in memory.

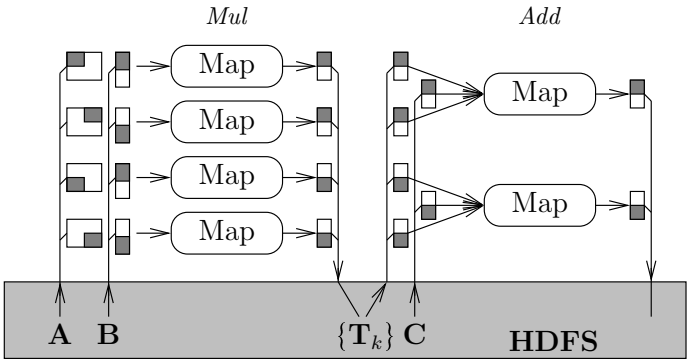

FIGURE 2.4: Cumulon's map-only Hadoop jobs for $\mathbf{A} \times \mathbf{B} + \mathbf{C}$ using the physical plan template of Figure 2.1b.

we first show, in Section 2.2.1, why MapReduce is a poor fit for this type of programs. In Section 2.2.2, after describing Cumulon's storage and execution models, we sample a couple of interesting physical operators designed for matrices, and show how to implement Cumulon conveniently on top of Hadoop (without adhering to MapReduce).

2.2.1 Why not MapReduce

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

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

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

overlapping subsets of input.

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

Example: Matrix Multiply Consider multiplying two big matrices $\mathbf{A}$ (of size $m \times l$) and $\mathbf{B}$ (of size $l \times n$). Let $\mathbf{C}$ denote the result (of size $m \times n$). On a high level, virtually all efficient parallel matrix multiply algorithms work by dividing input into submatrices: $\mathbf{A}$ into $f_m \times f_l$ submatrices, each with size $m/f_m \times l/f_l$, and $\mathbf{B}$ into $f_l \times f_n$ submatrices, each with size $l/f_l \times n/f_n$. We call f_m, f_l, and f_n *split factors*. The output will consist of $f_m \times f_n$ submatrices. Let $\mathbf{M}^{(i,j)}$ denote the submatrix of $\mathbf{M}$ at position (i, j). The output submatrix at position (i, j) is then computed as $\mathbf{C}^{(i,j)} = \sum_{k=1}^{f_l} \mathbf{A}^{(i,k)} \times \mathbf{B}^{(k,j)}$. Computation typically proceeds in two steps: first, pairs of submatrices from $\mathbf{A}$ and $\mathbf{B}$ are multiplied in parallel; second, the results of the multiplies are grouped (by the output submatrix position they contribute to) and summed. Note that if $f_l = 1$, the second step is not needed.

Indeed, matrix multiply is an example where one input submatrix may be needed by multiple tasks in the first step (unless $f_m = f_n = 1$, a special case that we will discuss later). If we adhere to MapReduce, we have to use the map phase to replicate the input submatrices and shuffle them to the reduce phase to be multiplied. Next, if $f_l \neq 1$, we need a second MapReduce job to perform the required summation, adding even more overhead. SystemML [14] takes this approach (called *RMM* there), but to avoid the second MapReduce job, it requires $f_l = 1$ (see Figure 2.2 for illustration).

However, SystemML's RMM turns out to be suboptimal on many accounts. First, the map phase performs no useful computation; input replication and shuffling are just expensive workarounds to allow reduce tasks to access data they need. Second, $f_l = 1$ severely limits the choice of execution strategies: it basically amounts to

multiplying rows of **A** with columns of **B**. It is well known in the literature on out-of-core linear algebra [41] that such an aspect ratio is suboptimal for matrix multiply (assuming sufficiently large m, l, and n). The number of element multiplications per task in this case is only linear in the number of input elements per task; in contrast, using square submatrices of size $k \times k$, for example, would yield k^3 multiplications per $2k^2$ input elements. Since the total number of element multiplications across all tasks is fixed ($m \times l \times n$), strategies that require fewer input elements for the same number of multiplications have lower I/O. We will experimentally confirm the suboptimality of SystemML's RMM strategy in Section 2.5.1.

In the other special case of $f_m = f_n = 1$, we basically multiply columns of **A** with rows of **B**: each submatrix multiply generates a matrix of the same size as the final output **C**, and these matrices simply need to be added to produce **C**. Note that in this case the submatrix multiply tasks work on disjoint input data, though each task needs to draw a specific pair of submatrices from **A** and **B**. If we relax the MapReduce model to allow custom input splitting, it would be possible to use a map phase to perform all submatrix multiplies, and a reduce phase to perform the final add. However, under a "pure" MapReduce model, we would need one full MapReduce job for the submatrix multiplies and another one for the add: the map phase in both jobs serves the sole purpose of routing appropriate data to reduce tasks, which still incurs considerable overhead. This two-job strategy for $f_m = f_n = 1$ is called *CPMM* in SystemML [14]. RMM and CPMM together are the two matrix multiply strategies supported by SystemML.

HAMA [36] supports arbitrary split factors for matrix multiply. It assumes that input has already been preprocessed for a MapReduce job such that each map task receives the two submatrices to be multiplied as one input data item. The result submatrices are then shuffled to reduce tasks to be added. Thus, HAMA carries out all computation in a matrix multiply in a single MapReduce job. However, the

preprocessing step would still take another MapReduce job to replicate the input submatrices.

2.2.2 The Cumulon Approach

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

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

Tile size is chosen to be large enough to amortize the overhead of storage and access, and to enable effective compression. However, they should not be so big that they limit execution options (e.g., split factors in matrix multiply). See Section 2.5 for the default setting in experiments. Within a tile, elements are stored in column-major order. Both sparse and dense formats are supported: the sparse format enumerates indices and values of non-zero elements, while the dense format simply stores all values.

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

Bulk Synchronous Parallel model [42] where data are only communicated through the global, distributed storage in the unit of tiles.

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

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

Physical Plan Templates and Operators As defined in Section 2.1, a *deployment plan* completely specifies how to execute a Cumulon program. The *physical plan template* component ($\mathcal{L}$) of the deployment plan specifies a workflow of jobs, where each job is a DAG of *physical operators*. Each physical operator has a list of parameters, whose settings (by the $\mathcal{O}$ component in the deployment plan) control their execution behavior. During job execution, each task executes an instance of the DAG. Similar to database query execution, the physical operators execute in a pipelined fashion through an iterator-based interface. To reduce the overhead of element-wise data passing, the unit of data passing among iterators is a tile (when appropriate). Physical operators can read and write tiles in the input and output splits (respectively) assigned to their tasks.

Example Recall the two physical plan templates for $\mathbf{A} \times \mathbf{B} + \mathbf{C}$ in Figure 2.1. Here we fill in more details. The *Mul* physical operator has split factors f_m, f_l, and f_n as its parameters (they are not the only ones; more discussion will follow shortly): set by $\mathcal{O}$, they control the dimensions of two submatrices being multiplied. *Mul* reads these submatrices directly from distributed storage.

The physical plan template in Figure 2.1a works for $f_l = 1$, where *Mul* produces a submatrix of $\mathbf{A} \times \mathbf{B}$ directly (no summation is required), and passes this submatrix to *Add* in the same job (one tile at a time) to be added to the corresponding submatrix of $\mathbf{C}$.

The physical plan template in Figure 2.1b works in the general case of $f_l > 1$ and requires two jobs. The first job multiplies submatrices from $\mathbf{A}$ and $\mathbf{B}$, and writes f_l intermediate matrices, where the k-th intermediate matrix $\mathbf{T}_k$ consists of submatrics $\mathbf{T}_k^{(i,j)} = \mathbf{A}^{(i,k)} \times \mathbf{B}^{(k,j)}$; each instance of *Mul* is responsible for one submatrix in one of the intermediate matrices. The second job computes $(\sum_{k=1}^{f_l} \mathbf{T}_k) + \mathbf{C}$; each instance of *Add* reads corresponding tiles from $\mathbf{T}_1, \ldots, \mathbf{T}_k, \mathbf{C}$, adds them, and writes an output tile.

A Closer Look at Mul So far, our discussion of *Mul* has remained at a high level; next, we discuss how two input submatrices $\mathbf{A}^{(i,k)}$ and $\mathbf{B}^{(k,j)}$ are actually multiplied. There are many options, with different performance tradeoffs. We begin with two options illustrated in Figure 2.3. **1)** The most straightforward option is to read two submatrices entirely into memory and multiply them using a highly tuned BLAS library. This option has good CPU utilization but also high memory requirement. The amount of memory available to each slot limits the size of the submatrices that can be multiplied without thrashing. **2)** Another option, aimed at reducing memory requirement, is to read one submatrix in memory and stream in the other one in a specific order tile by tile. Say $\mathbf{A}^{(i,k)}$ is read and buffered in memory. Conceptually,

each column of output tiles is produced by multiplying $\mathbf{A}^{(i,k)}$ with each column of tiles in $\mathbf{B}^{(k,j)}$. To this end, we reserve memory for a column of output tiles, and stream in tiles of $\mathbf{B}^{(k,j)}$ in column-major order. The p-th tile in the current column of $\mathbf{B}^{(k,j)}$ would be multiplied with the p-th column of tiles in $\mathbf{A}^{(i,k)}$ and accumulated into the column of output tiles. While this option has a lower memory requirement, its multiplications are performed in a more piecemeal fashion, so it tends to result in lower CPU utilization than the first option.

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

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

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

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

As we will see in Section 2.5.1, *MaskMul* can improve performance dramatically. In particular, it speeds up Line 3 of PLSI (Algorithm 1) by more than an order of magnitude (Figure 2.9).

Need for Automatic Optimization From the discussion of physical operators, it is clear that we want automatic optimization of matrix-based data analysis programs. Even for simple programs with the most basic matrix operations, there are many choices to be made. For example, for Figure 2.1, should we choose two jobs, or one job with potentially suboptimal split factors? For matrix multiply, what are the best choices of split factors and streaming granularity? Depending on sparsity, should we use *Mul* or *MaskMul*? Some of these choices are related to each other, and to hardware provisioning and configuration settings that affect number of tasks (and split size per task), memory, CPU utilization, relative costs of CPU and I/O, etc. Making these choices manually may be infeasible, error-prone, or simply not cost-effective.

Implementation in Hadoop Our discussion so far only assumes generic storage and execution platforms. While this generality is by design—as we want Cumulon to be able to work with a variety of alternative platforms in the future—our current im-

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

plementation of Cumulon is based on Hadoop and HDFS, which allows us to readily leverage its features, flexibility, and ecosystem of tools and users. We now describe this implementation, highlighting how it departs from the traditional MapReduce-based usage of Hadoop.

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

Each Cumulon job is implemented as a *map-only* Hadoop job. Cumulon slots correspond to Hadoop map slots (we do not have reduce slots because we never use the reduce phase). Unlike MapReduce, a map task does not receive input as key-value pairs; instead, it simply gets specifications of the input splits it is responsible for, and reads directly from HDFS as needed. In-memory dense matrix multiplies call a (multi-threaded) JBLAS library. Each map task also writes all its output to HDFS; there is no shuffling.

For example, Figure 2.4 illustrates how Cumulon implements the physical plan template in Figure 2.1b with Hadoop and HDFS. Contrasting with SystemML's use of Hadoop in Figure 2.2, we see that Cumulon is simpler, more flexible (e.g., by supporting any choice of split factors), and has less overhead (by avoiding map tasks that do nothing but route and replicate data). These advantages will be confirmed by experiments in Section 2.5. Section 2.5.1 will also compare with another possible implementation of Cumulon on Hadoop that use reduce tasks (to get fewer jobs); we will see that the map-only approach described here works better.

2.3 Cost-Based Optimization

Overview We now turn to cost-based optimization of deployment plans for matrix-based data analysis in the cloud. Given a physical plan template $\mathcal{L}$—recall from Section 2.1 that Cumulon obtains possible $\mathcal{L}$'s from the original logical plan through rewrites and translation—we would like to find a deployment plan for $\mathcal{L}$ that is optimal in terms of completion time and total (monetary) cost. Recall that a deployment plan for $\mathcal{L}$ is a tuple $\langle \mathcal{L}, \mathcal{O}, \mathcal{P}, \mathcal{Q} \rangle$, where $\mathcal{O}$ represents the physical operator parameter settings, $\mathcal{P}$ represents the hardware provisioning settings, and $\mathcal{Q}$ represents the configuration settings. Let $\mathfrak{T}(\cdot)$ and $\mathfrak{C}(\cdot)$ denote the completion time and cost of a deployment plan, respectively. In this chapter, we focus on:

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

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

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

Hadoop and Amazon EC2 Currently, Cumulon is implemented on top of Hadoop and targets deployment on Amazon EC2. Under this setting, we can spell out the components $\mathcal{P}$ and $\mathcal{Q}$ of a deployment plan. $\mathcal{P}$ specifies, for each job, the type and number

(N) of machines to reserve on EC2; a change in either type or number of machines across adjacent jobs implies cluster switching. Ω specifies the Hadoop/HDFS configuration settings for each job. From our experience, the most relevant setting with a significant impact on plan performance is the number of map slots (S) per machine; we will therefore focus on this setting. Appropriate values for other settings can either be derived using S (such as memory available for each task) or chosen in ways independent from particular plans (such as HDFS block size).

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

Next, we discuss how to estimate the completion time and cost of a deployment plan (Section 2.3.1), and how to search the space of feasible plans for the optimal one (Section 2.3.2). Although much of the discussion will be specific to Hadoop and EC2, our technique should be applicable to other scenarios as well.

2.3.1 Time and Cost Estimation

On EC2, the total completion time and cost of a deployment plan come from the following sources: **1)** running Hadoop jobs; **2)** cluster initialization; **3)** cluster switching; **4)** data ingress/egress. (1) is the most significant source and will be the emphasis of this section. Time and cost for (2), (3), and (4) are relatively easier to estimate. (2) and (3) depend on the size and type of the clusters involved. (3) and (4) depend the amount of data transferred. For (3), the matrices that need to be copied to the new cluster can be obtained via live variable analysis. For (4), the cost depends on the method used for data ingress/egress. Currently we assume that all input/output

data of a program reside on Amazon EBS, but other methods and cost functions can be plugged in. For (2), (3), and (4), we will not go into additional details because of space constraints.

Since the cost of running a Hadoop job on EC2 can be readily calculated from the completion time, we will focus on estimating the expected completion time of a job in this section. Our overall approach is to build a model for job completion time using data collected from benchmarking and simulation. Since the number of possible settings of $\mathcal{L}$, $\mathcal{O}$, $\mathcal{P}$, and $\mathcal{Q}$ is huge, we cannot afford a single high-dimensional model; instead, we decompose it into components and build each component separately. In the remainder of subsection, we first discuss how to estimate completion time of individual tasks (Section 2.3.1) by modeling computation and network I/O separately, and then how to estimate job completion time from task completion time (Section 2.3.1).

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

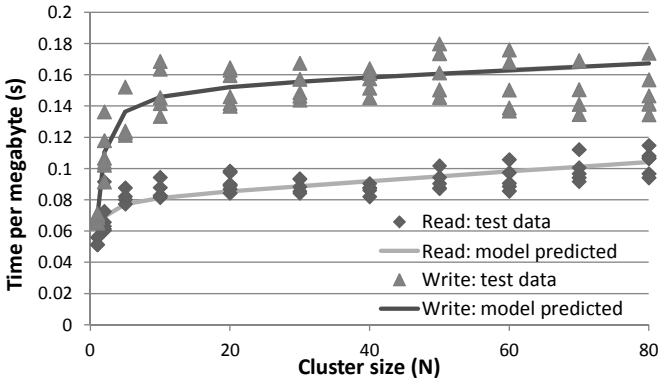

FIGURE 2.5: Estimated vs. actual times per unit I/O for machine type c1.medium with $S = 2$. The program measured here multiplies two dense matrices.

iterations; Section 2.4 discusses how Cumulon handles this challenge pragmatically.

Estimating Task Completion Time

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

Estimating Computation Time of a Task Recall that a task consists of a DAG of physical operators. We build a model of computation time for each operator and compose (add) them into a model for the task. A possible approach is to analytically derive a FLOP (floating-point operations) count model for each operator, but we have found that FLOP counts alone do not lead to an accurate estimate of computation time. For example, a JBLAS implementation of matrix multiply is easily

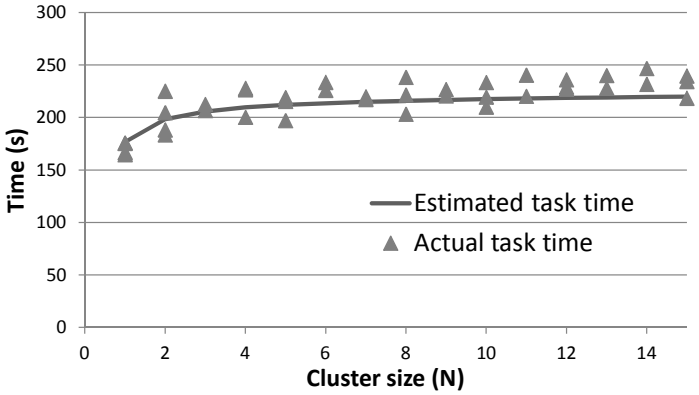

FIGURE 2.6: Estimated vs. actual average task completion times for machine type c1.medium with $S = 2$. The program here multiplies two 6k $\times$ 6k dense matrices.

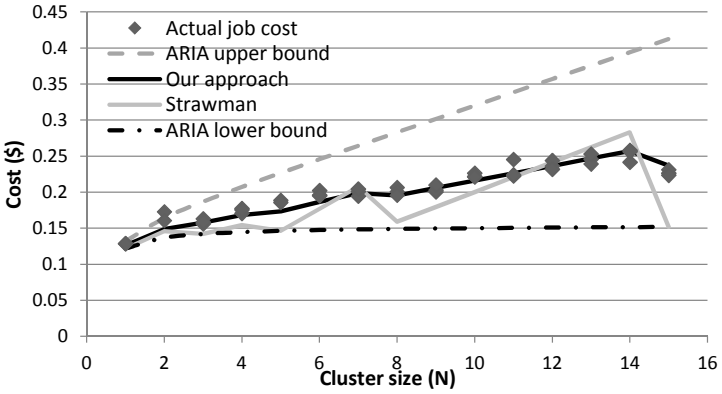

FIGURE 2.7: Actual job costs vs. predictions from the strawman approach, ours, as well as ARIA upper/lower bounds. The job multiplies two dense matrices of sizes 12k $\times$ 18k and 18k $\times$ 30k. Machine type is c1.medium; $q = 30$; $S = 2$.

orders-of-magnitude faster than hand-coded loops, despite the same FLOP counts; a sparse matrix multiply is much slower than a dense one with the same FLOP count. Therefore, we build a model for each operator by benchmarking.

In general, the model inputs include machine type, number of slots per machine, operator parameter settings, input split size, and input sparsity. There is no need to consider the number of machines in the cluster because computation is local. For some operators, the model can be pared down further. For example, for dense matrix *Mul*, we only need to benchmark for combinations of machine type, number of slots, and dimensions of sub-submatrices it chooses to multiply-accumulate each time using JBLAS. Then, the computation time of the whole *Mul* operator can be computed by multiplying the estimated time per JBLAS call by the number of such calls, which can be easily and analytically derived from the input split size, split factors, and granularity of streaming (recall Section 2.2.2).

Instead of benchmarking every possible setting of model inputs, we sample the space of feasible settings. For example, for *Mul*, machine type and number of slots limit the amount of memory available to each slot, and in turn bounds the sizes of sub-submatrices, which we additionally constrain to be integral multiples of tile sizes. The number of slots is also naturally limited on a given machine type; it makes no sense to allot a large number of slots on a machine with few cores because matrix computation is CPU-intensive. We repeat the benchmark for each setting multiple times; beside the mean, the sample variance will also be useful later (Section 2.3.1). Estimates for unseen settings are obtained using interpolation from neighboring benchmarked settings.

Though this approach is simple, we have found it very effective so far, both for highly predictable dense matrix operators, and for sparse matrix operators where the distribution of non-zero entries is random enough with respect to how input is split—this case happens, for example, to our PLSI program because when it generates $\mathbf{O}$,

it orders documents and words randomly (by their hashes). Future work is needed to better model the impact of skew in sparsity; new models can be readily incorporated into Cumulon.

Estimating Network I/O Time per Task Modeling of network I/O time is quite different from modeling computation. First, the average I/O speed depends on the size of the cluster (see Figure 2.5 for an example). Second, the total I/O volume is a reasonable predictor for I/O time (unlike FLOP count for computation time). Therefore, we estimate the total I/O volume per task, model time per unit I/O, and multiply them to obtain the I/O time estimate.

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

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

Figure 2.5 compares our model predictions with actual times per unit of data read/written by a dense matrix multiply program. A total of 108 actual measurements are plotted (60 of which are from multiplying two 6k $\times$ 6k matrices, and 48 are from multiplying a 4k $\times$ 8k matrix with an 8k $\times$ 8k one). The read/write models are each trained using 22 data points (disjoint from the 108 above) obtained by benchmarking the multiplication of 6k $\times$ 6k matrices. The figure shows that our models are quite effective, with coefficients of determination 0.92 (for reads) and 0.91 (for writes).

Putting It Together To estimate the expected completion time of a task, we add our estimates for the three components: computation, I/O, and task initialization. Figure 2.6 shows our estimate to be fairly consistent with the actual average task completion time for dense matrix multiply, over a wide range of cluster sizes. In this figure, each of the 40 data points plotted represents the measured average task completion for a test run (not used for training models). To obtain the variance in task completion time (which will be needed in Section 2.3.1 for estimating expected job completion time), we assume that the three component times are uncorrelated, and simply add their variances. The variance for the I/O time component is derived by scaling the standard deviation (root of variance) in v^{-1} (in seconds per MB) by the I/O volume (in MB).

Estimating Job Completion Time

Before presenting our approach to estimating job completion time, we start by discussing a strawman approach and a previous approach by *ARIA* [43]. Let q denote the number of tasks in a job (which can be derived from the input split size and total input size), and let μ denote the mean task completion time. Note that the total number of slots available for executing these tasks is NS.

Strawman With the strawman approach, we simply compute the number of waves as $\lceil q/NS \rceil$, and multiply it by μ to get the expected job completion time. If this approach works, we should see, as depicted by the plot for strawman in Figure 2.7, lots of jaggedness in the completion times (and hence costs) of a given job when N varies. Here, since each task remains the same, its completion time changes little with N. As N increases, $\lceil q/NS \rceil$ does not decrease smoothly; it decreases only at specific "breakpoints" of N. Thus, by this model, job completion time dips only at these points and remains flat otherwise; cost dips at the same points but goes up (because we get charged by N) otherwise. However, the actual job completion time/cost behavior, also shown in Figure 2.7, is smoother.

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

ARIA ARIA [43] characterizes task completion times by the observed average, minimum, and maximum among past runs of the same exact task. From these quantities, ARIA analytically derives an lower bound and an upper bound for job completion time. Figure 2.7 shows how these bounds translate to bounds on job cost (here ARIA is given 100 observations). ARIA estimates the expected job time/cost to be the average of the lower/upper bounds. From Figure 2.7, we see that the bounds are quite loose; although their average matches the general trend in actual job costs, it is too smooth and fails to capture the variations—its root-mean-square error (relative to actual average job costs) is 0.0189 (compared with 0.0078 for our approach).

Also, ARIA's analytical bounds do not consider different custom scheduler behaviors, which are possible in Hadoop.

Our Approach To account for the effect of variance (denoted σ^2) in task completion times, Cumulon models job completion time as a function of q, NS, and μ as well as σ^2. Specifically, the function has form $\mu(\lceil q/NS \rceil + (\frac{q \mod NS}{NS} - \alpha(\frac{\sigma}{\mu})) \cdot \beta(\frac{\sigma}{\mu}))$. Intuitively, the number of waves is adjusted by a fraction that accounts for σ^2 and the scheduler behavior. We take a simulation-based approach to support custom scheduler behavior. Given q, NS, and σ/μ, we draw task completion times from a truncated Gaussian with $(1, (\sigma/\mu)^2)$, and simulate the scheduler to obtain the expected job time. Using simulation results, we fit $\alpha(\frac{\sigma}{\mu})$ and $\beta(\frac{\sigma}{\mu})$ as low-degree polynomials. As Figure 2.7 shows, this approach models the job costs best, capturing both general trends and local variations.

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

2.3.2 Search Strategy

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

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

As Section 2.5.2 will demonstrate with experiments, it is necessary to jointly consider operator parameter settings as well as hardware provisioning and configuration settings. The main challenge addressed in this section is how to cope with the large

search space.

Establishing the Range of Cluster Sizes The number of machines (N) is a parameter in $\mathcal{P}$ with a potentially large range. We can narrow down this range by making two observations. For a given job running on a cluster with a particular machine type: **1)** as we decrease N, the job's best possible expected completion time cannot decrease; **2)** we can find a function $\check{c}(N)$ that lower-bounds the expected cost of the job on a cluster of size N, such that $\check{c}(N)$ is non-decreasing in N. Both observations are intuitive. (1) states that a job cannot run slower with more resources. While not always true for extremely large values of N (where the overhead of distribution could make a job run slower), the observation will likely hold in the range of N we actually consider in practice (unless we are given unreasonably tight $T_{\max}$). As for (2), while cost may not always increase with N, the general trend is increasing (as exemplified by Figure 2.7) as the overhead of parallelization kicks in. Our model of job time in Section 2.3.1 allows us to derive $\check{c}(N+1)$ after searching the deployment plans for a cluster of size N (details omitted because of space constraints).

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

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

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

The innermost loop of $\mathsf{opt}_{\mathcal{O},\mathcal{P},\mathcal{Q}}$ calls $\mathsf{opt}_{\mathcal{O},\mathcal{Q}}$. Since $\mathsf{opt}_{\mathcal{O},\mathcal{Q}}$ is given a cluster of fixed type and size, $\mathsf{opt}_{\mathcal{O},\mathcal{Q}}$ can optimize each job $j \in \mathcal{L}$ independently. Currently, we use a straightforward method to find the optimal $\mathcal{O}$ and $\mathcal{Q}$ for each j. We loop through possible values of S (which are naturally capped by a small multiple of the number of cores per machine, because our programs are CPU-intensive), and, for each S, sample the space of feasible operator parameter settings such that the total memory required by j's operators does not exceed a slot's available memory. As mentioned earlier, we also extend this method to compute a cost lower bound for a cluster of the same type and size $N + 1$ (details omitted). Finally, $\mathsf{opt}_{\mathcal{O},\mathcal{Q}}$ returns the optimal $\mathcal{O}$ and $\mathcal{Q}$ for $\mathcal{L}$, which consists of the optimal settings for each job in $\mathcal{L}$. Although simple, this method is fast enough in practice, because each job tends not to have many operators (see Section 2.5.2 for some optimization time numbers). The modular design of Cumulon's optimization algorithm allows more sophisticated methods to be plugged in if needed.

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

make sense to switch clusters for these steps, but additional switching between other steps will unlikely result in significant overall improvement.

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

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

2.4 Other Issues

Iterative Programs For simplicity of presentation, we have not yet discussed loops. Cumulon supports a loop construct in the workflow of jobs. It is intended for iterative programs (not for writing low-level, C-style loops that implements matrix multiply, for example). As mentioned in Section 2.3.1, cost estimation for iterative programs is very difficult, because convergence depends on factors ranging from input characteristics (e.g., condition numbers of matrices) to starting conditions (e.g., warm vs. cold).

Cumulon takes a pragmatic approach in this regard. We focus on predicting the per-iteration time and cost of an iterative program. For short loop bodies (such as PLSI), we first unroll it to enable more optimization opportunities. The user can either reason with this per-iteration "rate" produced by Cumulon, or specify the desired number of iterations to get the overall cost. In practice, this limitation is not severe, because users often start with trial runs until they understand the

Algorithm 2: Optimizing deployment plan with no cluster switching.

$\textbf{opt}_{\mathcal{O},\mathcal{Q}}(\mathcal{L},\mathcal{P})$: $\mathcal{L}$: physical plan template;
$\mathcal{P}$: type and number of machines in the cluster.
returns: deployment plan with lowest cost for given $\mathcal{L}$ and $\mathcal{P}$;
if two return values are expected, the second lower-bounds the cost on larger clusters of the same type (details omitted).

$\mathcal{O}, \mathcal{Q} \leftarrow \varnothing, \varnothing$;
foreach *job* $j \in \mathcal{L}$ **do**
  $C_{\text{best}}, S_{\text{best}}, O_{\text{best}} = \infty, \perp, \perp$;
  foreach *reasonable slot number per machine* S **do**
    foreach *feasible setting* O *of* j*'s operator parameters* **do**
      if $\mathfrak{C}(\langle j, O, \mathcal{P}, (S)\rangle) < C_{best}$ **then**
        $C_{\text{best}}, S_{\text{best}}, O_{\text{best}} = \mathfrak{C}(\langle j, O, \mathcal{P}, (S)\rangle), S, O$;
  $\mathcal{O} \leftarrow \mathcal{O} \cup \{(j, O_{\text{best}})\}$; $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{(j, S_{\text{best}})\}$;
return $\langle \mathcal{L}, \mathcal{O}, \mathcal{P}, \mathcal{Q}\rangle$;

$\textbf{opt}_{\mathcal{O},\mathcal{P},\mathcal{Q}}(\mathcal{L}, T_{\textbf{max}})$: $\mathcal{L}$: physical plan template;
$T_{\max}$: maximum completion time.
returns: lowest-cost deployment plan for $\mathcal{L}$ that runs under $T_{\max}$.

$C_{\max}, D_{\text{best}} \leftarrow \infty, \perp$;
foreach *machine type* m **do**
  $N \leftarrow \min\{N \mid \mathfrak{C}(\mathsf{opt}_{\mathcal{O},\mathcal{Q}}(\mathcal{L}, (m, N)) \leqslant T_{\max}\}$ by exponential search;
  repeat
    $D, \check{c} \leftarrow \mathsf{opt}_{\mathcal{O},\mathcal{Q}}(\mathcal{L}, (m, N))$;
    if $\mathfrak{C}(D) < C_{max}$ **then** $C_{\max}, D_{\text{best}} \leftarrow \mathfrak{C}(D), D$;
    ;
  until $\check{c} \geqslant C_{max}$;
return D_{best};

convergence properties on full datasets. An interesting direction of future work is to look for cases where we can do more, e.g., when factors affecting convergence are understood and can be efficiently computed without running the program.

Coping with Performance Variance In Section 2.3.1, we have shown how to deal with variance in task completion time, which can result from transient behaviors or self-interference among tasks, and affect our prediction of expected job completion time. But this step is only a beginning. Performance uncertainty can arise for many reasons. It can stem from our optimizer's incomplete or inaccurate knowledge of the

input properties: e.g., whether and where non-zero entries are clustered in a sparse matrix. In a cloud setting, for a (virtual) machine of given type, a cloud provider may assign an AMD instead of an Intel CPU, which affects the performance of linear algebra libraries. Depending on where the provisioned machines are located in the provider's data center, the network characteristics may vary. Performance can also be impacted by other virtual machines running on the same physical machine, and by other workloads sharing the same physical network.

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

2.5 Experiments

Our experiments are conducted on Amazon EC2. We consider the EC2 machine types listed in Table 1.1, and use the published rates (but without rounding up usage hours). The HDFS block size is 32MB; Cumulon's tiles have the same size by default (equivalent of a 2048×2048 dense double matrix). HDFS replication factor is set to 3 or the cluster size (N), whichever is smaller. These configuration settings do impact performance and reliability, but unlike the number of slots per machine (S), their choices are far less dependent on particular Cumulon programs. We have found these settings to work well for all workloads we tested, and therefore we do not include them in $\mathbb{Q}$ for plan optimization.

Algorithm 3: The loop body of GNMF. Here, $\circ$ denotes element-wise multiply and $\circ\!\!/$ denotes element-wise divide.

Data: $\mathbf{V}$: $m \times n$ sparse data matrix.
Result: $\mathbf{W}$: $n \times k$ dense matrix;
$\mathbf{H}$: $k \times m$ dense matrix;
k is usually much less than m and n.

1 initialize $\mathbf{H}, \mathbf{W}$;
2 **repeat**
3 $\quad \mathbf{H}' \leftarrow \mathbf{H} \circ (\mathbf{W}^\intercal \times \mathbf{V}) \circ\!\!/ (\mathbf{W}^\intercal \times \mathbf{W} \times \mathbf{H})$;
4 $\quad \mathbf{W}' \leftarrow \mathbf{W} \circ (\mathbf{V} \times \mathbf{H}^\intercal) \circ\!\!/ (\mathbf{W} \times \mathbf{H} \times \mathbf{H}^\intercal)$;
5 $\quad \mathbf{H}, \mathbf{W} \leftarrow \mathbf{H}', \mathbf{W}'$;
6 **until** *termination condition*;

We use a variety of programs to evaluate Cumulon. 1) ***Matrix multiplies*** with varying dimensions and sparsity allow us to experiment with different input characteristics. 2) ***RSVD-1***, introduced in Chapter 1, is the dominating step in a randomized SVD algorithm. 3) ***GNMF-1*** is Line 3 in Algorithm 3, which describes a popular algorithm called *Gaussian Non-Negative Matrix Factorization* (*GNMF*) [30], with applications including vision, document clustering, and recommendation systems. It is also used by SystemML [14] for evaluation. 4) ***PLSI-1*** is Line 3 in Algorithm 1, an expensive step in a PLSI iteration. 5) ***PLSI-full*** is a Hadoop program consisting of a non-Cumulon part, which downloads and preprocesses documents into the word-document matrix $\mathbf{O}$, followed by a Cumulon part, which performs PLSI on $\mathbf{O}$.

2.5.1 Execution Engine and Physical Operators

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

Comparison with SystemML First, we compare Cumulon with SystemML using a number of matrix multiply programs with varying sizes, shape, and sparsity. Their

Table 2.1: Choices of split factors by SystemML and Cumulon for $\mathbf{A} \times \mathbf{B}$.

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

running times on the same cluster, broken down by jobs and phases, are shown in Figure 2.8. For SystemML, the number of map slots and that of reduce slots are always 2, and we choose the faster one of their RMM and CPMM strategies for each program. For Cumulon, we use the deployment plans picked by Cumulon; the number of (map) slots is also always 2. Table 2.1 shows the input dimensions and the split factors ended up being used by SystemML and Cumulon.

For Cumulon, we omit streaming granularity settings (Section 2.2.2) here and in subsequent discussion; in most cases choosing this granularity to be a tile works best, as CPU under-utilization can be compensated with a large enough S.

From Figure 2.8, we see that Cumulon is significantly faster than SystemML (and hence proportionally cheaper in terms of monetary cost), for all but one program (#4). For #4, the two inputs are small and shaped like column and row vectors, where SystemML's RMM strategy happens to use the same (optimal) split factors as Cumulon; thus the two have comparable performance.

Two factors attribute to Cumulon's advantage: 1) Cumulon's map tasks do more useful work than SystemML's; 2) SystemML's choices of split factors are severely limited. We examine these factors in isolation later when discussing Figures 2.2 and 2.10.

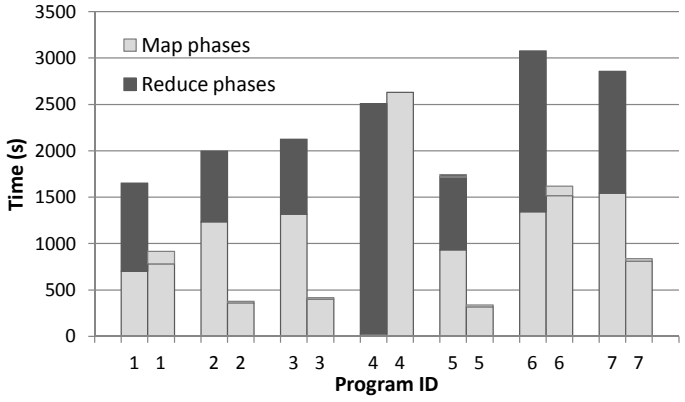

FIGURE 2.8: SystemML vs. Cumulon for matrix multiply programs (see table in text), on 10 m1.large machines. For each program, the left bar shows SystemML and the right bar shows Cumulon.

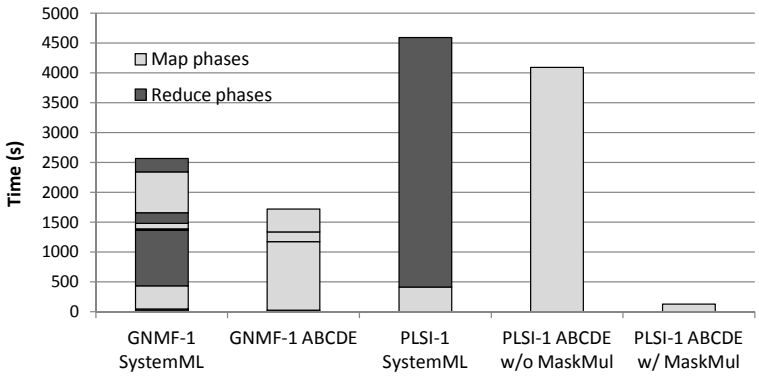

FIGURE 2.9: SystemML vs. Cumulon for GNMF-1 and PLSI-1, on 10 m1.large machines (some jobs/phases are short and thus may be difficult to discern from the figure).

Table 2.2: Options for multiplying two dense 48k × 48k matrices using flexible split factors, on 10 c1.xlarge machines. († Implementation-dependent; irrelevant here as pure MapReduce's first job alone is already more costly than other options.)

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

More Complex Workflows and Advantage of MaskMul Figure 2.9 compares SystemML and Cumulon on two workflows, GNMF-1 and PLSI-1, on the same cluster. For PLSI-1, $m = 3375104$, $n = 131151$, $k = 100$. GNMF-1 computes $\mathbf{H} \circ (\mathbf{H}^\intercal \times \mathbf{V}) \oslash (\mathbf{W}^\intercal \times \mathbf{W} \times \mathbf{H})$, where $\mathbf{H}$ is $k \times m$, $\mathbf{V}$ is $n \times m$, and $\mathbf{W}$ is $n \times k$ as in PLSI-1. There are two map slots per machine, and for SystemML, also two reduce slots. SystemML uses 5 full MapReduce jobs for GNMF-1, and 1 for PLSI-1; in contrast, Cumulon uses 4 and 1 *map-only* jobs, respectively. From Figure 2.9, we see that advantages of Cumulon carry over to more complex workflows.

Figure 2.9 further compares two plans for PLSI-1 in Cumulon: one that does not use the *MaskMul* operator (recall Section 2.2.2) and one that uses it. We see that performance improvement enabled by *MaskMul* is dramatic (more than an order of magnitude), further widening the lead over SystemML.[3]

Comparison with Other Hadoop-Based Implementations Going beyond SystemML, we examine more broadly how to implement matrix multiply with general split factors on top of Hadoop. We compare three options. 1) Cumulon: Two map-only jobs, where the first multiplies and the second adds. 2) ***Pure MapReduce***: Two map-reduce jobs, where the map tasks in the first job simply replicate and shuffle input

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

to reduce tasks to be multiplied, like in SystemML. Unlike SystemML, however, flexible choices of split factors are allowed, so the second job may be required for adding. (This option enables us to study the impact of different uses of map tasks in isolation from the effect of different split factors.) 3) ***Cumulon+reduce***: One full map-reduce job, where the map tasks read directly from HDFS to multiply, like Cumulon; however, the results are shuffled to reduce tasks to be added.

Figure 2.2 compares the running times of these options in the same cluster, as well as settings of split factors and number of slots (optimized for respective options). First, we see that pure MapReduce incurs a significant amount of pure overhead in its map tasks; in contrast, other options' map tasks perform all multiplications. Second, the map-only approach of Cumulon allows it to focus all resources on map tasks, without the overhead and interference caused by having to shuffle to reduce tasks. For example, because of resource contention, Cumulon+reduce is forced to allocate 3 map slots for multiplication (vs. 4 for Cumulon) and use bigger split factors (which lead to more I/O). Also, general sorting-based shuffling supported by Hadoop is an overkill for routing data to matrix operators, who know exactly which tiles they need. Thus, in spite of the fact that writes to HDFS are slower than to local disks (for shuffling), Cumulon's approach of passing data through HDFS between map-only jobs is the better option.

2.5.2 Deployment Plan Space and Optimization

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

Impact of Operator Parameter and Configuration Settings We show that even if we are given a fixed cluster, different $\mathcal{O}$ and $\mathcal{Q}$ settings can have a big impact on performance. Figure 2.10 compares a number of deployment plans for a dense matrix multiply on the same cluster, while the number of slots per machine (S) varies. For each S setting, we consider the plan with the optimal $\mathcal{O}$ (split factors) for this S, as well as a plan with same split factors as SystemML's RMM (which requires $f_l = 1$). However, this latter plan is implemented in Cumulon and does not carry the overhead of SystemML's pure MapReduce implementation. (Thus, this plan helps reveal the impact of SystemML's suboptimal split factor choices in isolation from the overhead of its restricted use of map tasks.)

From Figure 2.10 we make two observations. First, poor choices of $\mathcal{O}$ and $\mathcal{P}$ lead to poor performance; e.g., the second plan for $S = 1$ is nearly three times more expensive than the optimal plan with $S = 6$. Second, the choices of $\mathcal{O}$ and $\mathcal{P}$ are interrelated. As S increases, memory available to each slot decreases, which obviously limits the input split size. However, the effect of S on the optimal split factors is more intricate than simply scaling them proportionally to make splits fit in memory, as we can see from Figure 2.10.

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

Impact of Machine Type Next, we see how machine types influence optimality of deployment plans. Figure 2.11 explores various plans for dense matrix multiply on a 20-machine cluster of four types. There are four "base" plans, each optimized for one type. As the legend shows, they have rather different $\mathcal{O}$ and $\mathcal{Q}$ settings.

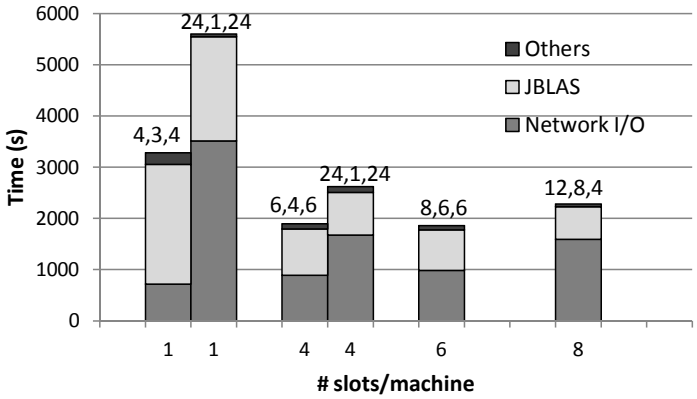

FIGURE 2.10: Effect of operator parameter and configuration settings (split factors and number of slots per machine) on running time of multiplying two 48k $\times$ 48k dense matrices. The cluster has 10 c1.xlarge machines. The split factors for *Mul* are shown on top of the respective bars.

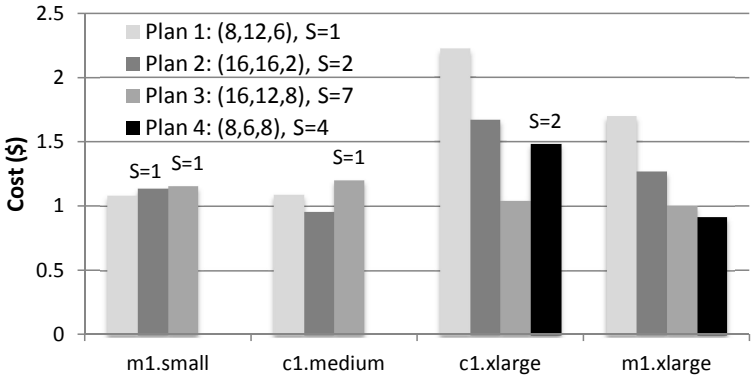

FIGURE 2.11: Effect of machine type on the cost of multiplying two 96k $\times$ 96k dense matrices. $N = 20$. The legend shows, for each base plan, its settings of the three split factors and the number (S) of slots per machine. Adjustment to S (if needed) is shown on top of the respective bar.

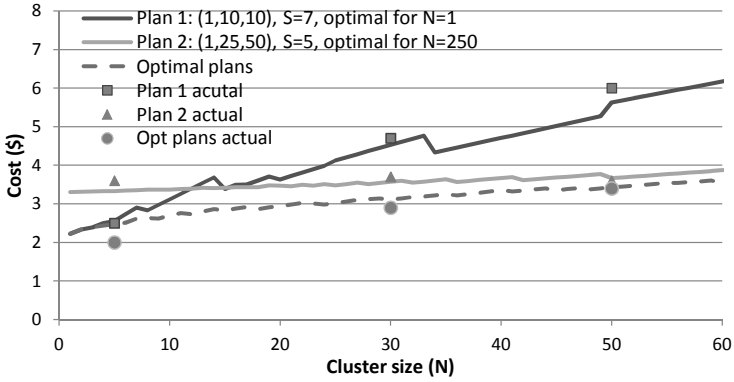

FIGURE 2.12: Effect of cluster size on the cost of RSVD-1 (with $l = 2\text{k}, m = 200\text{k}, n = 200\text{k}, k = 5$) on c1.xlarge machines. Cost curves are predicted; costs of actual runs for sample data points are shown for comparison.

Figure 2.11 also shows what happens if one simply "retools" a plan optimized for one machine type for another. In particular, if the target machine type does not have enough memory run this plan, we lower the plan's S setting so that it can run without thrashing. This retooling is not always feasible; e.g., Plan 4, optimized for m1.xlarge, cannot be retooled for m1.small and c1.medium. In general, retooled plans are suboptimal, and in some cases (e.g., Plan 1 retooled for c1.xlarge) can incur twice as much cost as the optimal.[4]

Impact of Cluster Size Even if we fix the machine type, optimal settings for $\mathcal{O}$ and Ω will change as we vary the cluster size N. In Figure 2.12, we consider RSVD-1 on increasing large clusters of c1.xlarge machines. First, we find Plan 1 as the best for $N = 1$, and Plan 2 as the best for $N = 250$. Their $\mathcal{O}$ and Ω settings are shown in

[4] Figure 2.11 shows monetary cost, not time. As a side note on how to interpret such figures, having the lowest cost (Plan 2 and c1.medium in this case) may not imply the best deployment plan in the presence of tighter completion time constraints. Here, Plans 3 and 4 actually complete quicker, but are costlier because the rates of their machine types are higher.

the legend. We retool the plans for different cluster sizes in a straightforward way: they simply run with their $\mathcal{O}$ and $\mathcal{Q}$ settings, so work done by each task remains the same, and varying N translates to varying number of waves. The predicted costs of running these plans retooled from Plans 1 and 2 are plotted in Figure 2.12. For comparison, the figure also plots the predicted cost of the optimal plan found by our optimizer for every N.

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

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

Putting It Together We now show how Cumulon can help users make intelligent deployment decisions by generating Figure 1.1 seen earlier in Chapter 1. The figure visually presents the tradeoff between completion time and monetary cost[5] across different machine types, and enables users to ask "what-if" questions with different deadline and budget scenarios. Each data point of a cost curve is obtained by invoking the Cumulon optimizer to find the lowest-cost deployment plan for RSVD-1 that completes within the given time for the given machine type. This plan specifies not only parameter settings for its physical operators but also the cluster size and

[5] Cost of data ingress/ogress depends on the setup, and is not shown here. Assuming Amazon EBS for stable storage, this cost would be no more than $2 (same for all deployment plans in Figure 1.1).

the number of slots per cluster. On a desktop machine with moderate hardware, it took under a minute to generate the whole figure (excluding the costs of actual runs for sample data points). This optimization overhead is well worth the benefit provided by Figure 1.1—one actual run can easily take hours to complete.

To see that Cumulon's cost estimation is effective, we also conducted actual runs for some data points; the results are largely in line with the predictions. Additional evaluation of cost estimation can be found in Figure 2.12 and figures in Section 2.3.1.

Cluster Switching and PLSI-Full Finding a simple program that warrants cluster switching turned out to be difficult. Cluster switching did not bring significant benefit to matrix multiplies, RSVD-1, GNMF-1, or PLSI-1, or complete versions of the last three algorithms. For example, in complete RSVD, while cluster switching at the end of RSVD-1 may seem obvious, it actually makes little difference because so little work remains after this dominating RSVD-1 step. Hence, our experience confirms our conjecture in Section 2.3.2 and justifies our approach of prioritizing the search for plans with few or no switches.

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

Finally, we consider PLSI-full, which includes both non-matrix and matrix parts.

Table 2.3: Detailed comparison of three plans for PLSI-full.

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

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

It turns out that while the first part of PLSI-full prefers a large number of small machines, the second part prefers fewer but larger machines. Thus, we select two candidate clusters: C_1, with 20 c1.medium machines, and C_2, with 6 m1.xlarge machines. We consider three plans: 1) run the entire PLSI-full on C_1; 2) run the entire PLSI-full on C_2; 3) run the first part on C_1, switch to C_2, and run the second part. The results are summarized in Table 2.3. As we can see, cluster switching reduces the cost from $22.62 to $20.59.

2.6 Related Work

A number of recent projects supporting statistical computing on big data subject scientists and statisticians to new, and in most cases, lower-level, programming abstractions. RHIPE [15], Ricardo [11], and Analytics' R/Hadoop enable R code to invoke Hadoop and vice versa, but programmers are responsible for parallelization and tuning. Apache Mahout and MADlib [10, 17] offer libraries of statistical analysis routines (for Hadoop and a parallel database system, respectively). Historically,

library-based approaches have been either limited by hard-coded assumptions tied to particular data representations and execution environments, or ridden with knobs that are difficult to tune for new deployment settings.

Automatic, cost-based optimization is a staple of database systems. Indeed, MADlib leverages a database optimizer to alleviate tuning burden, but database optimizers are often inadequate for matrix-based workloads. Database-style optimization has also been applied to general MapReduce systems, e.g., by Starfish [18]. However, such work faces difficult challenges in modeling arbitrary map and reduce functions. Because Cumulon focuses on matrix-based programs, we are able to take a white-box approach specialized for such workloads. SystemML [14] takes an approach similar to ours, but as we have seen earlier, Cumulon avoids some significant overheads that SystemML incurs with its MapReduce-based implementation. Furthermore, SystemML only considers optimization of execution, while Cumulon also jointly considers hardware provisioning and configuration settings.

Automatic resource provisioning has been studied recently for general MapReduce systems [27, 43, 51, 19]. In contrast to the black-box workloads that these systems face, we focus on declaratively specified statistical workloads, which lead us to a different challenges and opportunities as we have shown in this chapter.

On a high level, we share the goal of making big-data analytics easier with many recent systems. For example, projects such as SciDB [6] are rebuilding entire systems from the ground up for specific workload types. Targeting graph mining, PEGASUS [28] implements *generalized iterated matrix-vector multiply* efficiently on Hadoop. RIOT [50] and RevoScaleR focus on making statistical computing workloads in R I/O-efficient; pR [31] automatically parallelizes function calls and loops in R. Pig [34], Hive [40], and SciHadoop [8] are examples of higher-level languages and execution plan generators for MapReduce systems. Our work goes beyond these systems by addressing important usability issues of automatic hardware provisioning

and configuration.

2.7 Conclusion and Future Work

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

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

3

Cümülön v1

3.1 Introduction

In this chapter, we will discuss Cümülön v1, the first Cumulon verison with auction-based market support. Amazon EC2 is a leader that provides an auction-based market of computing resources. As we mentioned in Chapter 1, 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. The user cannot change the bid price once the bid is placed.

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 prices 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. Furthermore, 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

[1] Amazon EC2's pricing scheme is actually more nuanced and can lead to some rather interesting bidding strategies; more details are in Section 3.3.2.

use them? Is there any way to bound this uncertainty while still saving some cost in the expected sense?

Our Contributions In this chapter, we present our answers to the challenges above in the context of *Cümülön v1* (i.e., *Cumulon* with spots). As mentioned in Chapter 1, Cümülön v1 starts with an optimal "baseline" plan, that only uses on-demand instances, and that meets the user-specified completion deadline and minimizes expected monetary cost. Next, Cümülön v1 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.[2] We also show how to build a highly elastic computation and storage engine for matrices on top of spot instances.

A key desideratum of Cümülön v1 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 v1 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 v1 how much risk she is willing to take while trying to reduce the expected cost.

Cümülön v1 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 v1 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

[2] As we will show in Section 3.3.6, since the execution time of Cümülön v1's plan is almost always shorter than the estimated completion time of the baseline plan regardless of spot price uncertainty, the user-specified deadline (which is no shorter) will be met with high probability. This is why Cümülön v1 focuses on minimizing cost.

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 v1 uses a combination of (implicit) caching and (explicit) *sync* operations to copy selected states from the transient store to the primary one. Taking advantage of declarative program specification (with matrices and linear algebra), Cümülön v1 uses fine-grained lineage information to minimize the work required to recover lost data.

In terms of optimization support for spot instances, Cümülön v1 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 v1 estimates execution time 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 v1 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 v1 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 chapter assumes a modified version of Amazon's pricing policy (see Section 3.3.2 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 chapter, we describe the inner workings of Cümülön v1, evaluate how well Cümülön v1 supports the use of spot instances, and explore how the pricing policy affects bidding strategies.

3.2 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 v1 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 here.

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.

3.2.1 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 prevent data loss [32], but storing all intermediate results in it rather than local disk is prohibitively inefficient. **2)** Using a *single HDFS on top of both primary and transient nodes* is a natural extension to

Cumulon 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 was done in [9], is cheap and reliable. Tasks on transient 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 3.4). Other storage approaches will be discussed later in related works in Section 3.5.

As mentioned in Chapter 1, Cümülön v1 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 (cache) 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.[3] To enable this optimization, Cümülön v1 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 ob-

[3] 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 3.2.3).

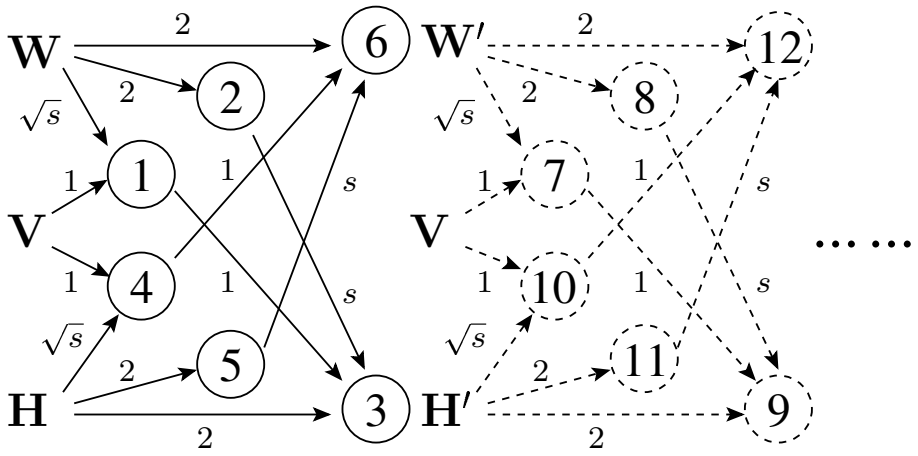

FIGURE 3.1: Dependencies among jobs for two iterations of GNMF. Edges are labeled with read factors, where s denotes the number of tasks per job.

tained at optimization time as a function of the split size by analyzing j's physical plan template. During a job i, Cümülön v1 sets the "cacheable" bit for $\mathbf{A}$ to 1 iff $\sum_{j \geqslant i} \gamma_j^{\mathbf{A}} > 1$. To illustrate, consider the following example.

Figure 3.1 shows the dependencies among jobs and matrices for two GNMF iterations. Cümülön v1 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, and each submatrix of $\mathbf{W}$ multiples with n' submatrices of $\mathbf{V}$. Therefore, each submatrix of $\mathbf{W}$ is needed by n' tasks, so the read factor (of $\mathbf{W}$ in job 1) $\gamma_1^{\mathbf{W}} = n'$, or the square root of the number of tasks in the job.

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 in the

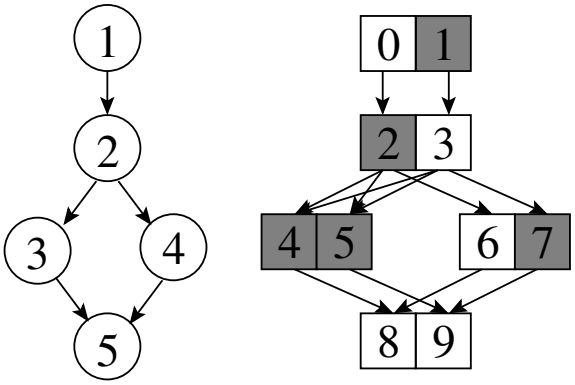

FIGURE 3.2: A toy workflow with 5 jobs (left) and the lineage among their corresponding output tiles (right). Jobs are shown as circles. Tiles are shown as squares; shaded tiles are lost in a hit.

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 v1 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 v1 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.

3.2.2 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, could 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. Two specialized physical operators are developed for a *sync* job, *SyncRead* and *SyncWrite*. Rather than reading all tiles in the input split from either home store or non-home store, *SyncRead* reads the set of tiles in the matrix present *only* in the transient store by consulting the tile manager. [4] *SyncWrite* always writes to the primary store, rather than the home store. As with other jobs, this *sync* 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 **A** will be used later, how likely a hit will occur before the uses of **A**, how much of **A** will be cached by the primary store over time, and how expensive it is to recompute the part of **A** required when recovering from a hit. Some of these factors can be determined by static program analysis; some further depend on the bid price and future market prices. It would be unrealistic to expect users to come up good syncing strategies manually. Therefore, Cümülön v1 considers the choice of a syncing strategy as an integral part of its optimization (Section 3.3).

3.2.3 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 v1 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 v1 does not rely on having any complete snapshot of the execution state. Thanks to its knowledge of the physical plan template, Cümülön v1 is able to redo a job "partially," using tasks

[4] In the case of syncing the result matrix of a matrix multiplication job, this involves reading the collection of partial result tiles from both stores and summing them into tiles in the final result matrix.

that may differ from those in the original execution.

To help determine the dependencies between input and output data, each physical operator in Cümülön v1 supports a *lineage function*, which returns the subset of input tiles required for computing a given subset of output tiles. The lineage function of a job is simply the composition of 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 $\Lambda^{\mathbf{A}}_{\mathbf{B}}(\mathbb{B})$ to denote the subset of $\mathbf{A}$ tiles required for computing the given subset $\mathbb{B}$ of $\mathbf{B}$ tiles, as determined by the lineage function of the job.

Once Cümülön v1 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[5] or time running out). Next, Cümülön v1 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 recursively. 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 part of the final output or 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 $j+1, j+2, \ldots, j_{\mathsf{hit}}$ that reads $\mathbf{A}$. To recover the deficient tiles for each $\mathbf{B} \in \mathcal{B}$, we need the subset $\Lambda^{\mathbf{A}}_{\mathbf{B}}(X^{\mathbf{B}})$ of $\mathbf{A}$'s tiles. Therefore, $X^{\mathbf{A}} = \bigcup_{\mathbf{B}\in\mathcal{B}} \Lambda^{\mathbf{A}}_{\mathbf{B}}(X^{\mathbf{B}}) \backslash P^{\mathbf{A}}$.

Then, Cümülön v1 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 $j_{\mathsf{hit}} + 1$.

[5] In fact, with lineage information, it is possible for Cümülön v1 to infer which tasks will fail due to missing data, without waiting for them to time out. Cümülön v1 currently does not implement this feature, however.

As a toy example, Figure 3.2 shows a workflow with the tile-level lineage it generates. Jobs are numbered according to execution order, and every output matrix consists of two tiles. If a hit occurs during the execution of job 4 and the shaded tiles are lost, then we need the full output of jobs 3 and 4 (i.e., tiles 4–7) in order to finish the workflow. After lineage analysis, Cümülön v1 will know that tiles 2, 4, 5 and 7 are deficient and must be regenerated. Note that although tile 1 is also missing, it is not deficient because tile 3 is available. The recovery plan is as follows: run job 2 partially to generate tile 2 from tile 0, then job 3 in full to generate tiles 4 and 5, and finally job 4 to generate tile 7 from tile 3.

Note that the division of work in a partial job into tasks can be quite different from the original execution; the recovery plan will use parameters optimized for execution on primary nodes only, as opposed to those optimized for the full-strength cluster. In other words, Cümülön v1 performs recovery in a more flexible, data-driven manner than just redoing a subset of the original tasks. Furthermore, Cümülön v1 does not track lineage explicitly, but instead infer it as needed from the physical plan template. Such features are possible because Cümülön v1 programs are specified declaratively using a vocabulary of operators with known semantics. These features distinguish Cümülön v1 from other systems with lineage-based recovery (such as *Spark* [48]) that need to support black-box computation.

3.3 Optimization

There is a huge space of alternatives for running a Cümülön v1 program with transient nodes—from execution to bidding to syncing. Moreover, Cümülön v1 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 opti-

mization problem to consider more complex plans. Specifically, we start with the following restrictions:

(Augmenting a Baseline Plan Template) 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. This baseline plan (involving no transient nodes) can be the lowest-cost plan found by Cumulon, under a user-specified deadline.

Let Q denote the program's physical plan template, and let q_j denote the physical plan template of job j, where $1 \leqslant j \leqslant m$ and m is the number of jobs. We only consider plans that *augment* Q with transient nodes: 1) we will not change the set of primary nodes; 2) we will not change Q, except for adding *sync* jobs. However, we do reoptimize the execution parameters for Q—e.g., splits for each job (recall Section 2.2)—for the new cluster.

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 type as the primary ones at the start of the program. Suppose the market price of the transient nodes at bid time is p_0. The bidding strategy is characterized by $(\hat{p}, n_{\text{tran}})$, where $\hat{p} \geqslant p_0$ is the bid price and $n_{\text{tran}} \geqslant 0$

is the number of transient nodes to bid for.

As discussed in Section 3.2, Cümülön v1 in fact has full system support for heterogeneous clusters. However, optimization becomes considerably more complex; we are 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 $\mathfrak{T}_{\mathsf{hit}}$.
- 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 $\mathfrak{T}_{\mathsf{rec}}$.
- 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 $\mathfrak{T}_{\mathsf{rap}}$.

In general, as mentioned earlier in Footnote 2, with a very high probability that the augmented plan runs faster than the estimated time of the baseline plan. This is

because in all three execution phases (except in a *sync* job), the primary nodes are consistently doing non-repeating useful work with all results preserved. In a *sync* job, results are being preserved into primary store at a even higher rate than normal execution. On the other hand, no matter when the hit happens, the amount of work done by the transient nodes and preserved in primary store will be non-negative. They only helps reducing execution time.

There are extreme situations where the augmented plan ends up slightly slower than the baseline plan. For instance, the impact of egress read traffic from primary store (discussed later in Section 3.3.3) could slow down the execution in primary nodes when the transient nodes initially join execution. If the hit happens early when no work done in transient nodes is preserved, we could end up with a execution time slightly longer than baseline plan. However, this happens with a very small probability in general. Visualizations on execution time will be provided later in Section 3.3.6.

Since the execution time of the augmented plan is very likely shorter than that of the baseline plan, the user-specified deadline (which is no shorter) will be met with high probability. That's why Cümülön v1 focuses on minimizing cost when searching for augmented plans. In this chapter, 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 $(\hat{p}, n_{\text{tran}})$ and syncing strategy S that minimize the expected cost of the 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 3.3.9.

For the remainder of this section, we begin with the market price model in Section 3.3.1 and pricing scheme in Section 3.3.2, which let us estimate $\mathfrak{T}_{\mathsf{hit}}$, and calculate costs given cluster composition and lengths of the execution phases. We present models for estimating execution times in Sections 3.3.3 and 3.3.4. We then show how to combine these models to obtain the total cost distribution (with uncertainty) in Section 3.3.5, and how to solve the optimization problem in Section 3.3.6.

3.3.1 Market Price Model

In order to estimate $\mathfrak{T}_{\mathsf{hit}}$ 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 v1 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 3.4.3), and then combine the components to create a sample path. Further technical details on the market price modeling together with its validation are organized into Appendix B at the end of the thesis.

Although our full model captures periodicity, we instead use a simpler, non-periodic price model by default in our experiments for better interpretability of results, unless otherwise specified. 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. In Section 3.4.5, the full periodic model will be used to provide more insights into the optimal bid time.

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 $\mathfrak{T}_{\mathsf{hit}}$. For example, the top part of Figure 3.8 (ignore the bottom for now) shows the distribution of $\mathfrak{T}_{\mathsf{hit}}$ 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 $1 - P(\mathfrak{T}_{\mathsf{hit}} \leqslant 2h) = 0.32$.

3.3.2 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 ($\mathfrak{T}_{\mathsf{hit}}$, $\mathfrak{T}_{\mathsf{rec}}$, and $\mathfrak{T}_{\mathsf{rap}}$), the total cost is

$$\begin{aligned} &C(n_{\mathsf{prim}}, n_{\mathsf{tran}}, \mathfrak{T}_{\mathsf{hit}}, \mathfrak{T}_{\mathsf{rec}}, \mathfrak{T}_{\mathsf{rap}}) \\ &= n_{\mathsf{prim}} p_{\mathsf{prim}} (\mathfrak{T}_{\mathsf{hit}} + \mathfrak{T}_{\mathsf{rec}} + \mathfrak{T}_{\mathsf{rap}}) + n_{\mathsf{tran}} \int_0^{\mathfrak{T}_{\mathsf{hit}}} p(t)\, dt. \end{aligned} \tag{3.1}$$

Basically, the primary nodes are charged at the constant rate of p_{prim} throughout 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 reclaimed. 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 (as Microsoft Azure and Google Cloud do). Cümülön v1 can support the Amazon scheme (or any other alternative) if needed. In fact, in Section 3.4.4, we will investigate how optimal strategies change as we switch to the Amazon scheme, or another plausible pricing scheme where transient nodes are charged at their bidding prices.

3.3.3 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 [21]). 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

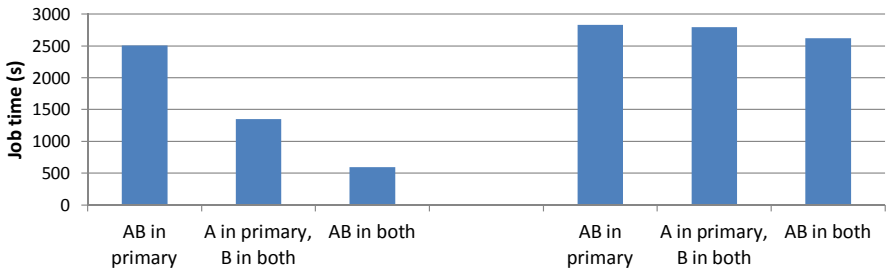

FIGURE 3.3: Job time of $A + B$ ($120k$ square dense, left three columns) and $A \times B$ ($60k$ square dense, right three columns) with different inital locations in a 6+40 c1.meidum cluster. Here k denotes 1024.

where all nodes participate in the distributed store and are expected to be available throughout the execution.

Cümülön v1 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.

For instance, Figure 3.3 shows the job execution time of two jobs, $\mathbf{A} + \mathbf{B}$ and $\mathbf{A} \times \mathbf{B}$, in a 6+40 (6 primary nodes and 40 transient nodes) c1.medium cluster with two slots per node. Each task does a $4k \times 4k$ addition or a $6k \times 6k \times 6k$ multiplication, where k denotes 1024. "$\mathbf{A}, \mathbf{B}$ in both" is the case where both matrices are loaded in both primary and transient store beforehand, and thus serves as the base case. If $\mathbf{A}$ starts only in primary, then job time increases because more reading requests goes to primary store, creating a potential I/O bottleneck. If both $\mathbf{A}$ and $\mathbf{B}$ start

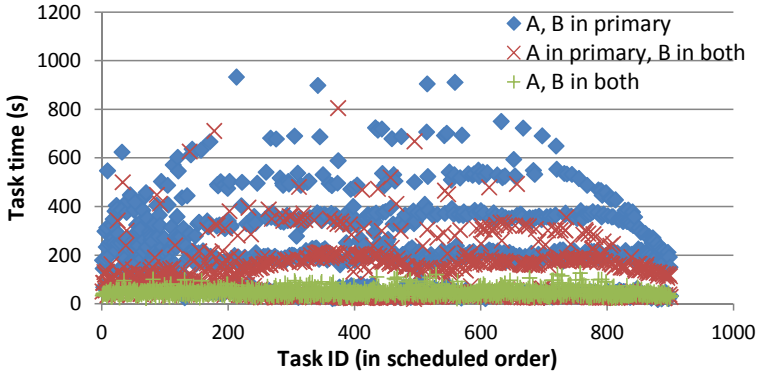

FIGURE 3.4: Task time of the $\mathbf{A} + \mathbf{B}$ job in Figure 3.3.

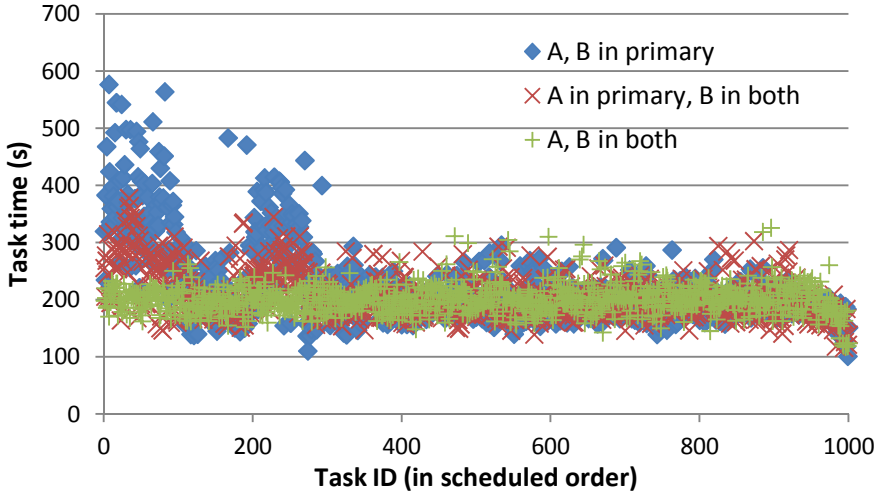

FIGURE 3.5: Task time of the $\mathbf{A} \times \mathbf{B}$ job in Figure 3.3.

only in primary, the job time delay doubled comparing to the previous case, since the amount of data to read from primary doubled.

As we can see, the bottleneck of reading from primary store has a larger impact on job time in the addition job than the multiplication. This is because in the addition job, computation is trivial and almost all job time is spent on I/O. While in the multiply job, computation takes a decent portion of fixed time. That is why the

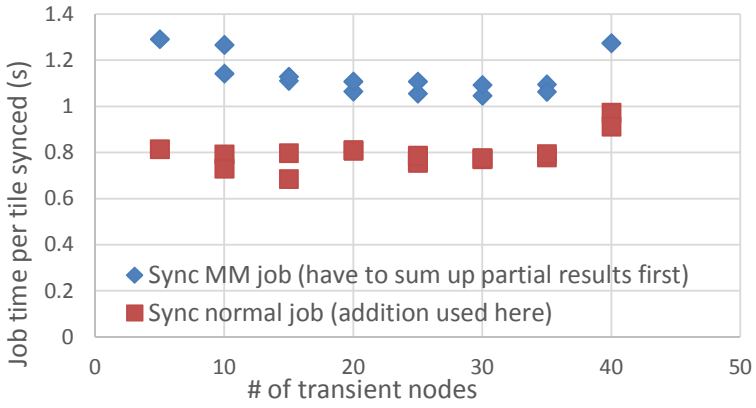

FIGURE 3.6: Sync-to-primary job time per tile synced.

impact of I/O contention on total job time is less significant.

On the other hand, since each matrix tile are needed by multiple tasks in the matrix multiply job, without data caching and sharing among transient nodes, the I/O bottleneck of the primary store could potentially be much worse than in addition, where each tile is only needed by exactly one task. Figure 3.4 and Figure 3.5 gives a task level timing for both jobs. In the addition job, when data are initally only in primary, all tasks in the job are experiencing I/O contention because every task is reading different data from primary. While in multiply, we only see significant delay in the first few waves (92 tasks per wave), and later on the work proceeds the same as the base case where both matrices are loaded into both stores. The transient cache works well in this case.

Figure 3.6 shows the sync-to-primary job time divided by the number of tiles synced using 6 c1.medium primary nodes. We can see that the average time to sync one tile does not change much when the transient size varies from 5 to 40. In the sync MM job case, the tiles to sync is the result of the previous matrix multiplication job, which is not summed up yet. In this *sync* job, each task have to read in all the partial

tiles from both primary and transient store, sum them up and write the final result tile to primary. The addtional reads done per tile is the reason why the time spent on syncing one tile is more than the normal job case. In conclusion, this figure shows that the bandwidth approach of estimating the influence of this kind of unbalanced traffic is effective in considerable large range of primary-to-transient ratio.

To account for these I/O patterns, Cümülön v1 extends the I/O time model for the two cases above with two additional terms: both have the form of *total unbalanced I/O amount / primary store bandwidth*; one is for data egress (reads) while the other is for data ingress (writes). Cümülön v1 adds a weighted sum of these two terms to the job time predicted by Cumulon; we train the weights and measure the per-node bandwidths for each machine type and for each physical operator using micro-benchmarks. Except for the two I/O patterns above, Cümülön v1 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 $\mathfrak{T}(n_{\mathsf{prim}}, n_{\mathsf{tran}}, q)$ denote the estimated completion time of the optimal job plan in the given cluster (recall the optimization in Section 2.3.1). The extended I/O time model is invoked 1) if $n_{\mathsf{tran}} > 0$ 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 v1 optimizer performs this estimation.

3.3.4 Sync and Recovery Time Estimation

While Section 3.3.3 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 3.2.2 and 3.2.3 that at run time, Cümülön v1 uses tile-level persistency and lineage infor-

mation 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 v1 makes strong independence and uniformity assumptions. While these assumptions certainly do not hold in general, they work well for Cümülön v1 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 $\gamma = \frac{n_{\mathsf{prim}}}{n_{\mathsf{prim}}+n_{\mathsf{tran}}}$. 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 3.2.1 that $\gamma^{\mathbf{A}}_j$ 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 $\rho^{\mathbf{A}}_{j,\phi}$ from $\rho^{\mathbf{A}}_{j-1,1}$ as follows. Because Cümülön v1 caches reads in its home store (Section 3.2.1), a read of $\mathbf{A}$ by a primary node can potentially increase the persistency of $\mathbf{A}$. For a tile to be absent from the primary store when job j is ϕ-completed, the tile must be absent before j (which happens with probability $1-\rho^{\mathbf{A}}_{j-1,1}$), and none of the $\phi\gamma^{\mathbf{A}}_j$ reads comes from a primary node (each of which happens with probability $1-\gamma$). Therefore, $\rho^{\mathbf{A}}_{j,\phi} = 1-(1-\rho^{\mathbf{A}}_{j-1,1})(1-\gamma)^{\phi\gamma^{\mathbf{A}}_j}$.

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

$$\widetilde{\mathfrak{T}^{\mathcal{A},j}_{sync}} = \mathfrak{T}\Big(n_{\mathsf{prim}}, n_{\mathsf{tran}}, sync\big(\sum_{\mathbf{A}\in\mathcal{A}}(1-\rho^{\mathbf{A}}_{j,1})\cdot \mathrm{size}(\mathbf{A})\big)\Big), \tag{3.2}$$

where $\mathfrak{T}(n_{\mathsf{prim}}, n_{\mathsf{tran}}, sync(v))$ estimates the time it takes for a *sync* job to persist v amount of tiles from the transient store to the primary store, using the I/O time model of Section 3.3.3. [6] Of course, if we *sync* $\mathbf{A}$ after job j, the persistency of $\mathbf{A}$ becomes 1 if the *sync* job completes, or $\rho^{\mathbf{A}}_{j,1} + (1 - \rho^{\mathbf{A}}_{j,1})\phi$ 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 3.2.3 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 3.2.3. Suppose the job producing matrix $\mathbf{B}$ uses matrix $\mathbf{A}$ as input. Let $\lambda^{\mathbf{A}}_{\mathbf{B}}(f)$ 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 $\lambda^{\mathbf{A}}_{\mathbf{B}}(\cdot)$ by simply sampling the results of the lineage function $\Lambda^{\mathbf{A}}_{\mathbf{B}}(\mathbb{B})$ with different subsets $\mathbb{B}$ of $\mathbf{B}$'s tiles. No execution of the job is needed.

As an example, Figure 3.7 plots the learned coverage function $\lambda^{\mathbf{A}}_{\mathbf{C}}(\cdot)$ for a matrix multiply job $\mathbf{C} = \mathbf{A} \times \mathbf{B}$. For comparison, we also plot test data points obtained from actual runs, by introducing hits at random times during the job, and counting the fraction of missing output tiles and the fraction of $\mathbf{A}$ tiles required for computing them. We see that the coverage function here is nonlinear: we need 80% of $\mathbf{A}$ tiles when 10% of the output tiles are missing, and almost all of $\mathbf{A}$ when 30% of the output tiles are missing.

Calculation of deficiencies follows a procedure similar to that of determining the set of deficient tiles in Section 3.2.3. We start with job j_{hit} and work backwards.

[6] In the case of syncing the result matrix of a matrix multiplication job, as discussed in Footnote 4, additional read traffic will be modeled accordingly.

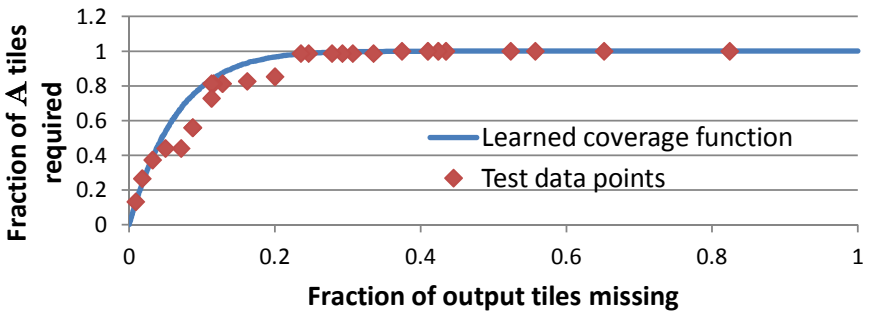

FIGURE 3.7: Coverage function $\lambda^{\mathbf{A}}_{\mathbf{C}}(f)$ for a matrix multiply job $\mathbf{C} = \mathbf{A} \times \mathbf{B}$, where $\mathbf{A}$ and $\mathbf{B}$ are square matrices with 225 tiles each.

Consider $\mathbf{A}$ produced by job j. If $\mathbf{A}$ is part of the final output or needed by any job following j_{hit}, $\chi^{\mathbf{A}} = 1 - \rho^{\mathbf{A}}_{j_{\text{hit}},\phi}$. Otherwise, let $\mathcal{B}$ denote the set of matrices produced by any job among $j+1, j+2, \ldots, j_{\text{hit}}$ that reads $\mathbf{A}$. For each $\mathbf{B} \in \mathcal{B}$, the fraction of $\mathbf{A}$ tiles that is needed in producing a $\chi^{\mathbf{B}}$ fraction of $\mathbf{B}$ tiles is $\lambda^{\mathbf{A}}_{\mathbf{B}}(\chi^{\mathbf{B}})$. Assuming independence, we have $\chi^{\mathbf{A}} = \left(1 - \prod_{\mathbf{B}\in\mathcal{B}}(1 - \lambda^{\mathbf{A}}_{\mathbf{B}}(\chi^{\mathbf{B}}))\right)(1 - \rho^{\mathbf{A}}_{j_{\text{hit}},\phi})$.

With deficiencies calculated, we can now estimate the execution time of the recovery phase. We need to run a partial version of job j ($1 \leqslant j \leqslant j_{\text{hit}}$) 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 $1 - \prod_{\mathbf{A}\in\mathcal{O}_i}(1 - \chi^{\mathbf{A}})$. Therefore, we can estimate the total execution time of the recovery phase as

$$\widetilde{\mathfrak{T}}_{\text{rec}} = \sum_{1\leqslant j\leqslant j_{\text{hit}}} \mathfrak{T}(n_{\text{prim}}, 0, q_j) \cdot \left(1 - \prod_{\mathbf{A}\in\mathcal{O}_j}(1 - \chi^{\mathbf{A}})\right), \tag{3.3}$$

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

3.3.5 Putting It Together: Cost Estimation

All components are now in place for us to describe Cümülön v1'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 v1 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 $t_1 \leqslant t'_1 < t_2 \leqslant t'_2 < \cdots < t_m$, where t_j is the estimated time when 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:

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

2. Given p_0, we simulate a market price trace $p(t)$ up to $t = t_m$ using the market price model. If $\forall t \in [0, t_m) : p(t) \leqslant \hat{p}$, we have "lucky" run without a hit, so $\mathfrak{T}_{\mathsf{hit}} = t_m$ and $\mathfrak{T}_{\mathsf{rec}} = \mathfrak{T}_{\mathsf{rap}} = 0$. Otherwise, we estimate $\mathfrak{T}_{\mathsf{hit}}$, $\mathfrak{T}_{\mathsf{rec}}$, and $\mathfrak{T}_{\mathsf{rap}}$ as follows:

   - $\mathfrak{T}_{\mathsf{hit}} = \min\{t \in [0, t_m) \mid p(t) > \hat{p}\}$.
   - $j_{\mathsf{hit}} = \max\{j \in [1, t_m] \mid t'_j < \mathfrak{T}_{\mathsf{hit}}\}$.
   - We estimate $\mathfrak{T}_{\mathsf{rec}}$ using Eq. (3.3). There are two cases: 1) If $\mathfrak{T}_{\mathsf{hit}} < t_{j_{\mathsf{hit}}}$, the cluster is in the middle of executing job j_{hit} when hit. We set ϕ for job j_{hit}

to $(\mathfrak{T}_{\mathsf{hit}} - t'_{j_{\mathsf{hit}}-1})/(t_{j_{\mathsf{hit}}} - t'_{j_{\mathsf{hit}}-1})$. 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 $(\mathfrak{T}_{\mathsf{hit}} - t_{j_{\mathsf{hit}}})/(t'_{j_{\mathsf{hit}}} - t_{j_{\mathsf{hit}}})$.

- We estimate $\mathfrak{T}_{\mathsf{rap}}$ as $\sum_{j\in[j_{\mathsf{hit}}+1,m]} \mathfrak{T}(n_{\mathsf{prim}}, 0, q_j)$.

Finally, we obtain the cost from Eq. (3.1) for the given price trace, using the estimated $\mathfrak{T}_{\mathsf{hit}}$, $\mathfrak{T}_{\mathsf{rec}}$, $\mathfrak{T}_{\mathsf{rap}}$, and $\int_0^{\mathfrak{T}_{\mathsf{hit}}} p(t)\,dt$ calculated from the 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 v1 does not account for uncertainty in the estimates of job execution times or data persistency/deficiency. For the linear algebra workloads targeted by Cümülön v1, 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.

3.3.6 Putting It Together: Optimization

To choose a bidding strategy, Cümülön v1 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, once $\hat{p}$ is high enough, additional increase in it will have very little impact—the distribution of $\mathfrak{T}_{\mathsf{hit}}$ 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 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, 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 $n_{\mathsf{tran}} = 300$ 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 $(n_{\mathsf{prim}}, n_{\mathsf{tran}})$, Cümülön v1 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 v1 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 v1 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. To recap, the two phases have goals matching the two criteria of our optimization problem: the first phase greedily reduces the expected cost, and the second phase then greedily lowers the risk. The complexity of this greedy algorithm is only quadratic in m.

To illustrate Cümülön v1'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 split sizes (recall Section 2.2), 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.

MM_γ^k Syncing Strategies In this example, we examine how Cümülön v1 chooses different syncing strategies based on a number of factors. First, we consider the distribution of $\mathfrak{T}_{\mathsf{hit}}$ predicted by our market price model, given $p_0 = \$0.02$ and $\hat{p} = \$0.2$. As explained earlier in Section 3.3.1, the top part of Figure 3.8 shows this distribution. Figure 3.8c plots the expected cost of MM_1^5 conditioned on $\mathfrak{T}_{\mathsf{hit}}$, for three plans that differ only in their syncing strategies; $n_{\mathsf{prim}} = 3$ and $n_{\mathsf{tran}} = 10$. We make the following observations.

- For every syncing strategy, its plot always starts with the baseline cost ($\mathfrak{T}_{\mathsf{hit}} = 0$); 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

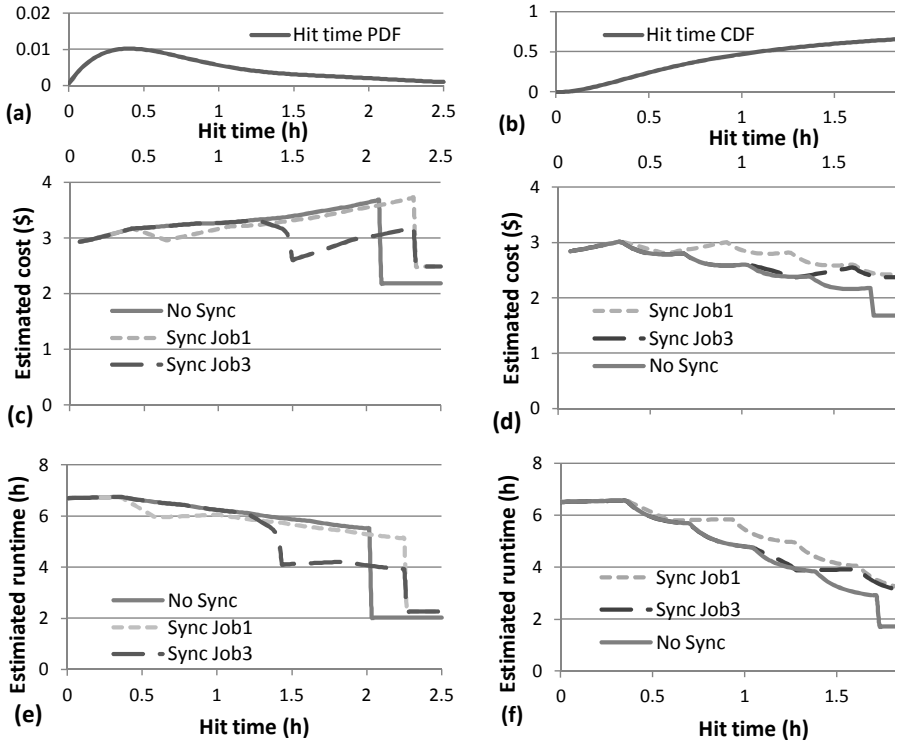

FIGURE 3.8: **Top:** Distribution of the hit time $\mathfrak{T}_{\text{hit}}$, with $p_0 = \$0.02$ and $\hat{p} = \$0.2$. a) PDF. b) CDF. **Middle/Bottom:** Expected total cost/runtime of MM^5_γ as a function of $\mathfrak{T}_{\text{hit}}$. c) e): Low read factor ($\gamma = 1$). d) f): High read factor ($\gamma = 5$).

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 3.8c that the cost and time of a lucky run are both higher than those 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 3.8c shows, syncing after the first multiply helps very little with cost,

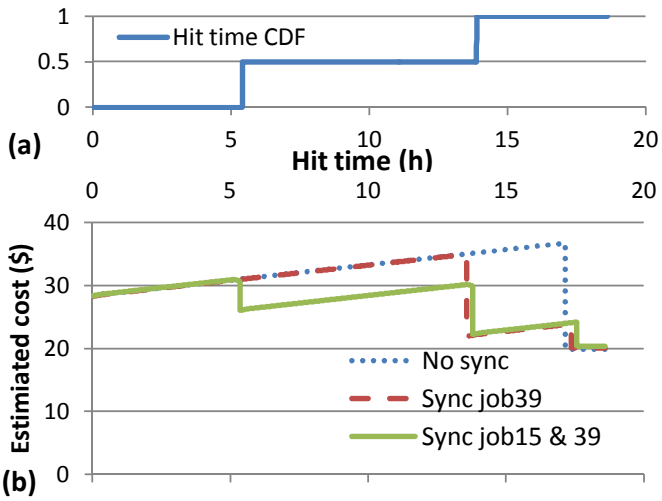

FIGURE 3.9: **Top:** CDF of a contrived $\mathfrak{T}_{\text{hit}}$ distribution, with price peaking at two known time points. **Bottom:** Expected total cost of MM_1^{50} as a function of $\mathfrak{T}_{\text{hit}}$.

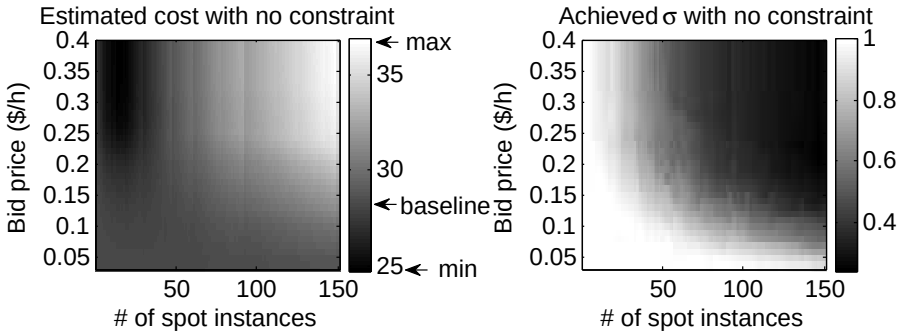

FIGURE 3.10: Optimal plans for MM_1^{50}, as we vary the bidding strategy. *No risk tolerance constraint* is set here. a) Intensity shows the estimated expected cost of the plan (darker is better). b) Intensity shows the probability that the cost is within 1.05 of the baseline (lighter is better).

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 v1 in this case 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.[7] Figure 3.8d 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 be recomputed during recovery. Thanks to this caching effect, as shown in Figure 3.8d, 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 3.3.3) as well as read factors and persistency (Section 3.3.4), Cümülön v1 is able to choose the strategy of no syncing intelligently.

Additionally, Figure 3.8e and f show the estimated total execution time ($\mathfrak{T}_{\text{hit}} + \mathfrak{T}_{\text{rec}} + \mathfrak{T}_{\text{rap}}$) of the two workflows, under various hit time. As discussed earlier in Section 3.3, using transient nodes in general helps reducing total execution time. The longer we hold the transient nodes, more useful work done by them will get persisted, leaving less work for the primary nodes, and thus less total execution time.

Next, we turn to the effect of future market price on syncing strategies. Consider the longer program MM_1^{50}. Figure 3.9a shows the distribution of $\mathfrak{T}_{\text{hit}}$ 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 3.9b 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 v1's greedy algorithm adds first the 39th, then the

[7] 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.

15th, before returning the result syncing strategy as its decision. From Figure 3.9b, we see that the two chosen *sync* jobs are timed to occur immediately before the two possible hit time points, which makes intuitive sense.

MM_1^{50} Plan Space Recall RSVD-1 in Chapter 1 and Figure 1.2. To see how different programs call for different plans, we now consider MM_1^{50} for comparison.[8] 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. Again, assuming $p_0 = \$0.02$, 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 v1 simply looks for plans with the lowest expected cost. Figure 3.10a plots the expected plan cost, while Figure 3.10b plots the probability that the cost stays within 1.05 of the baseline.

Figure 3.10 has a very different plan space compared with Figure 1.2. 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 3.10a. Also, larger clusters incur higher risks of cost overrun, as evidenced in Figure 3.10b.

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.

[8] 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 **A**'s square submatrices. This choice of splits is in fact suboptimal, and the optimal setting is described later when $\gamma = 5$.

3.3.7 Extension 1: Delayed Bidding

As discussed at the beginning of Section 3.3, we have made a number of assumptions in Cümülön v1 to make the optimization manageable. Now we introduce *delayed bidding* by relaxing the assumption on bid time (denoted by $\mathfrak{T}_{\mathsf{bid}}$).

In *delayed bidding*, rather than bidding at the start of the execution (denote the time then by $\mathfrak{T}_0$), we wait until $\mathfrak{T}_{\mathsf{bid}}$ (e.g. off-peak hours where market prices are expected to be lower) to place the bid for transient nodes. Execution starts with only primary nodes at $\mathfrak{T}_0$. We call it the *pre-bid* phase, whose duration is $\mathfrak{T}_{\mathsf{bid}}$. After the bid at time $\mathfrak{T}_{\mathsf{bid}}$, transient nodes will join the ongoing execution. This is the *pre-hit* phase described ealier in this section, followed by *recovery* and *wrap-up* phases if we are hit. The total execution cost in Eq. (3.1) now becomes:

$$\begin{aligned} &C(n_{\mathsf{prim}}, n_{\mathsf{tran}}, \mathfrak{T}_{\mathsf{bid}}, \mathfrak{T}_{\mathsf{hit}}, \mathfrak{T}_{\mathsf{rec}}, \mathfrak{T}_{\mathsf{rap}}) \\ &= n_{\mathsf{prim}} p_{\mathsf{prim}} (\mathfrak{T}_{\mathsf{bid}} + \mathfrak{T}_{\mathsf{hit}} + \mathfrak{T}_{\mathsf{rec}} + \mathfrak{T}_{\mathsf{rap}}) + n_{\mathsf{tran}} \int_{\mathfrak{T}_{\mathsf{bid}}}^{\mathfrak{T}_{\mathsf{hit}}} p(t)\, dt. \end{aligned} \tag{3.4}$$

We extend the Cümülön v1 optimizer to find the optimal bid time $\mathfrak{T}_{\mathsf{bid}}$, besides bidding and syncing strategy, so that the overall expected total cost (to be defined later in Eq. 3.5) is minimized. Note that all optimization decisions are still statically made at $\mathfrak{T}_0$ based on information available at that time (e.g. market price p_0). Since the market price at bid time (denoted by p_{bid}) is unknown at $\mathfrak{T}_0$, the optimal bidding strategy $(\hat{p}, n_{\mathsf{tran}})$ and syncing strategy S to be deployed at $\mathfrak{T}_{\mathsf{bid}}$ will be specified as a function of p_{bid}.

For each given $(\mathfrak{T}_{\mathsf{bid}}, p_{\mathsf{bid}})$ pair, if we subtract the fixed cost of the *pre-bid* phase $n_{\mathsf{prim}} p_{\mathsf{prim}} \mathfrak{T}_{\mathsf{bid}}$ from Eq. 3.4, minimizing the rest of the costs is equivalent to the problem of minimzing the cost in Eq. 3.1. We can treat the unexecuted part of the workflow at time $\mathfrak{T}_{\mathsf{bid}}$ as a new workflow and feed $p_0 = p_{\mathsf{bid}}$ into the optimizer presented in Section 3.3.6. Specifically, if the executing job (denoted by job j_{bid}) at time $\mathfrak{T}_{\mathsf{bid}}$ is

ϕ_0-completed, then the first job in the new workflow will have the same physical plan template, but with estimated time $\mathfrak{T}(n_{\mathsf{prim}}, n_{\mathsf{tran}}, q_1) \cdot (1 - \phi_0)$. Suppose $\mathbf{A}$ is produced by job j_{bid}, then its persistency will be $\rho^{\mathbf{A}}_{1,\phi} = (1 - \phi_0)\phi\gamma + \phi_0$ for the first job in the new workflow. Estimiation models derived in earlier sections can be directly applied on all later unexecuted jobs in the new workflow.

Let $\mathfrak{T}_{\mathsf{prim}}$ denote the execution time of the baseline plan of no bidding. The optimal decision on $\mathfrak{T}_{\mathsf{bid}}$ is made in two steps:

- For each possible combination of $(\mathfrak{T}_{\mathsf{bid}}, p_{\mathsf{bid}})$, $0 \leqslant \mathfrak{T}_{\mathsf{bid}} < \mathfrak{T}_{\mathsf{prim}}$, Cümülön v1 will search for an optimal bidding and syncing strategy that minimizes the expected cost in Eq. 3.4 and satisfies the risk constraint. Note that this plan always exists, even if the market price at bid time (p_{bid}) is very high. This is because the baseline plan itself (with $n_{\mathsf{tran}} = 0$) always satisfies the risk constraint. Denote this minimal expected cost found as $\mathfrak{C}(n_{\mathsf{prim}}, \mathfrak{T}_{\mathsf{bid}}, p_{\mathsf{bid}})$.

- Starting from the current market price p_0, a market price model (described in Section 3.3.1) will provide us a distribution of future market price at any time. Let $\mathbb{P}(p, \mathfrak{T})$ denote the probability for the market price to be p at time $\mathfrak{T}$, we have $\sum_p \mathbb{P}(p, \mathfrak{T}) = 1$. Cümülön v1 simply return the $\mathfrak{T}_{\mathsf{bid}}$ that minimizes

$$\mathfrak{C}(n_{\mathsf{prim}}, \mathfrak{T}_{\mathsf{bid}}) = \sum_{p_{\mathsf{bid}}} \mathfrak{C}(n_{\mathsf{prim}}, \mathfrak{T}_{\mathsf{bid}}, p_{\mathsf{bid}}) \cdot \mathbb{P}(p_{\mathsf{bid}}, \mathfrak{T}_{\mathsf{bid}}). \tag{3.5}$$

Note that since each plan associated with $\mathfrak{C}(n_{\mathsf{prim}}, \mathfrak{T}_{\mathsf{bid}}, p_{\mathsf{bid}})$ satisfies the risk constraint, picking the $\mathfrak{T}_{\mathsf{bid}}$ that minimizes $\mathfrak{C}(n_{\mathsf{prim}}, \mathfrak{T}_{\mathsf{bid}})$ suffices.

Delayed bidding becomes particularly useful when applied with a time-aware periodic market price model (Section 3.3.1), with information regarding peak and off-peak hours encoded. The optimizer will the prefer to wait until the off-peak hours when the price is likely to stay low for longer durations and place the bid then. Experimental results on *delayed bidding* will be discussed later in Section 3.4.5.

3.3.8 Extension 2: Flexible Primary Size

So far we assume that the baseline plan (i.e. the number of primary nodes to start with) is chosen beforehand so that the estimated execution cost could be minimized while ensuring that the user-specified deadline can be met. Now what happens if we can choose a different baseline plan?

In this extension, the primary size n_{prim} is included into Cümülön v1's search space. However, only the primary sizes where the baseline plan will meet the deadline are considered, so that we can still focus on cost and use the same optimization objective described in Section 3.3. Since a larger primary cluster means shorter estimated completion time and larger cost (due to scaling overhead) for the baseline plan, only larger primary clusters will be considered. Baseline plans with smaller primary cluster will miss the deadline.

Formally, let $\widetilde{n_{\mathsf{prim}}}$ denote the optimal n_{prim} whose baseline plan minimizes cost while meeting the deadline. Cümülön v1 will then consider every n_{prim} that is equal or larger than $\widetilde{n_{\mathsf{prim}}}$, and find the optimal plan that minimizes the overall expected total cost while satisfying the risk constraint. Note the baseline cost used in this constraint will be consistently the cost of the baseline plan corresponding to $\widetilde{n_{\mathsf{prim}}}$, rather than a moving target.

Starting from $\widetilde{n_{\mathsf{prim}}}$, Cümülön v1 will keep increasing n_{prim}. In general, the impact mainly comes from two aspects:

- The primary store will have a higher I/O bandwidth. This means faster primary store I/O (e.g. *sync* job). Furthermore, more transient nodes can potentially be utilized without higher I/O overhead. As a result, initially increasing primary size might leads to better plans with cheaper expected cost.

- Given the transient cluster and the fixed amount of work in the workflow, a bigger primary cluster will leave less work for the cheaper transient nodes. The

benefit of cheaper transient nodes is less exploited. In other words, since the primary nodes are more expensive, the plan could be more expensive although completion time might decrease.

As we increase n_{prim}, the impact of the first aspect decreases, the second aspect will eventually dominate. Going further will only increase the expected cost and decrease the number of feasible plans with respect to the constraint. Experimental results on this will be discussed in Section 3.4.6.

In the future, if we also want to consider the baseline plans with smaller primary clusters that could miss the deadline, then the optimization constraint needs to be modified since the possibility of missing the deadline should be incorporated. For instance, the new risk constraint coule be:

$$\mathbb{P}(\mathfrak{T} > \mathfrak{T}_{max}) + w \times \mathbb{P}(\mathfrak{C} > \mathfrak{C}_{max}) < p \tag{3.6}$$

Where $\mathfrak{T}_{max}$, $\mathfrak{C}_{max}$, w and p are all specified by user. Note that the optimization techniques decribed in this chapter can be readily modified and applied to the new constraints. We will leave these to future work.

3.3.9 Discussions

Besides delayed bidding and flexible primary size, *sequential bidding* is another immediate extension that allows Cümülön v1 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.

Going beyond this extension, 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 v1 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.

3.4 Experiments

We conduct our experiments on Amazon EC2. As mentioned in Section 3.2.1, we implement the dual-store design using two separate HDFS instances, with default replication factor set to 3. For brevity, let an $(n_{\mathsf{prim}}, n_{\mathsf{tran}})$ 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** (Chapter 1), **MM** $\frac{\boldsymbol{k}}{\boldsymbol{\gamma}}$(Section 3.3.6), and **GNMF** (Algorithm 3). By default in **GNMF**, we used $k = 100$ and a 7510k $\times$ 640k word-document matrix for $\mathbf{V}$, which is converted from a 2.5GB wiki text corpus in the preprocessing step. This convertion is implemented as a MapReduce job. Since this preprocessing step takes text files as input, it is beyond the scope of Cümülön v1 and thus the optimizations discussed here. In specifying matrix sizes, "k" denotes 1024.

3.4.1 Storage Design and I/O Policy

Benefit of a Distributed Transient Store As discussed in Section 3.2.1, Cümülön v1 implements a distributed store over the transient nodes. For comparison, we also implemented a "local" alternative, where each transient node manages its own cache in its local disk. With this alternative, cache accesses are faster because they are local, but data sharing is limited to tasks running on the same node, leading to more misses that require reading from the primary store.

We compare the alternatives using MM_5^1 on a $(1, 20)$ `m1.small` cluster. The job

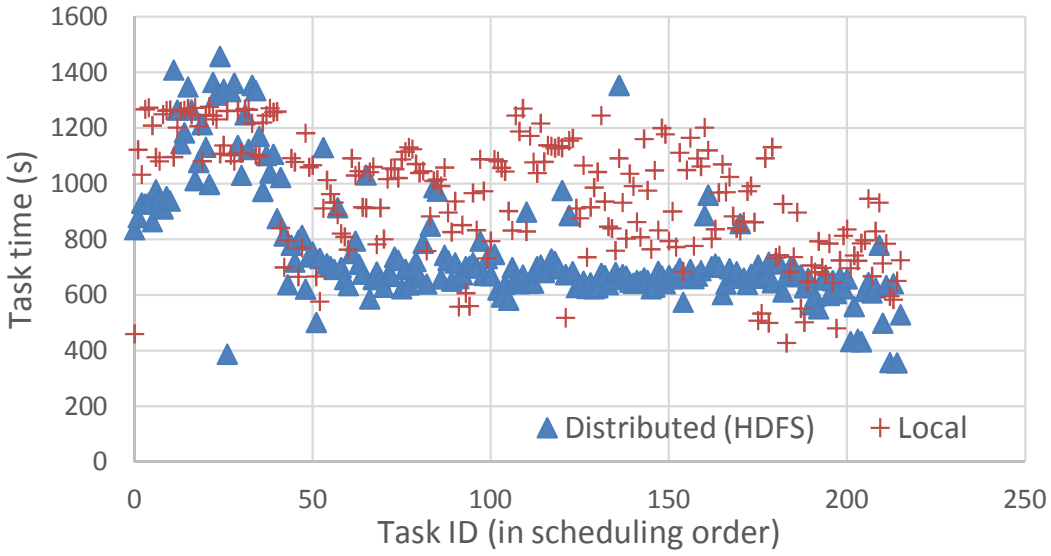

FIGURE 3.11: Task completion times in an MM_5^1 job, under Cümülön v1's distributed transient store vs. a local caching approach.

multiplies a 30k × 30k input matrix by itself. The input initially resides only on the primary node. The read factor of 5 is optimal under this setting. The transient nodes have two slots each, while the single primary node has one slot to mitigate the I/O bottleneck. For each alternative, Figure 3.11 plots the completion time for each task over the course of the matrix multiply job. Thanks to caching, task completion time decreases over time, because reading from the cache is faster. With Cümülön v1's distributed transient store, the drop is dramatic and occurs early—at the beginning of the second wave or task 42; at that point, a significant portion of the input becomes cached and shared among all transient nodes. In comparison, the benefit of the local caching alternative is gradual and much less overall.

I/O Policy We compare our dual-store design and I/O policy (Section 3.2.1) with three other alternatives. The baseline of comparison is *write primary + no read cache*, which is the third strawman solution described in Section 3.2.1. The two other alternatives can be seen as Cümülön v1's dual-store design with different features

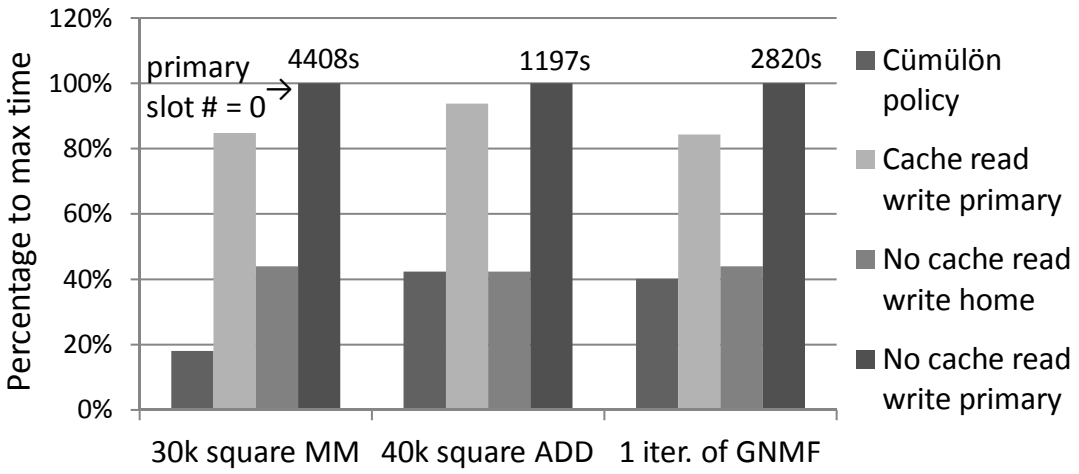

FIGURE 3.12: Comparison of alternative I/O policies for the dual-store design, across three workloads.

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 3.12 shows the execution times under the four I/O policies for three workloads with varying characteristics: MM_5^1, which multiplies two 30k $\times$ 30k input matrices; ADD, which adds two 40k $\times$ 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[9]). 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 3.12, we see that Cümülön v1'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

[9] 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.

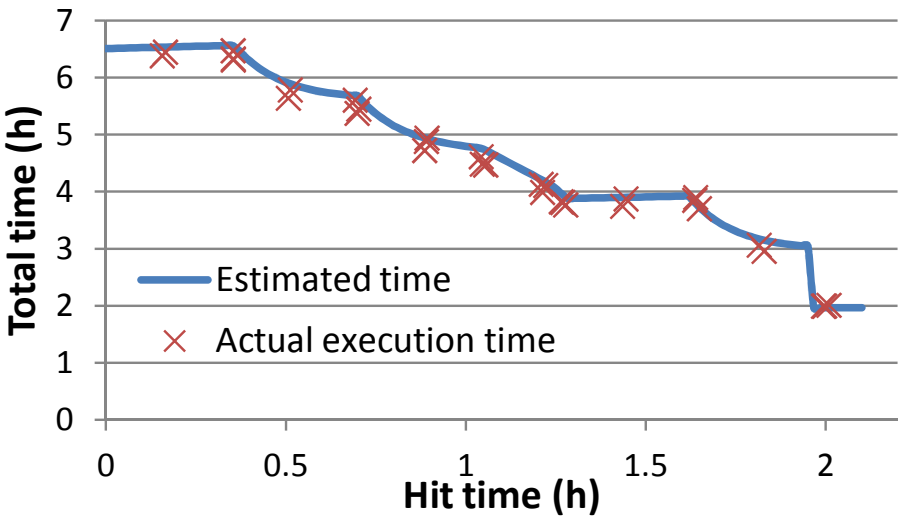

FIGURE 3.13: Estimated vs. actual total execution time of MM_5^5 as a hit occurs at different times.

v1 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 observation justifies Cümülön v1's design decision of not aggressively pushing writes across stores, but instead relying on caching and judicious use of *sync* jobs.

3.4.2 Time Estimation

We now turn to the validation of Cümülön v1'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.[10]

[10] Note that artificial hit injection gives us control over $\mathfrak{T}_{\text{hit}}$, 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.

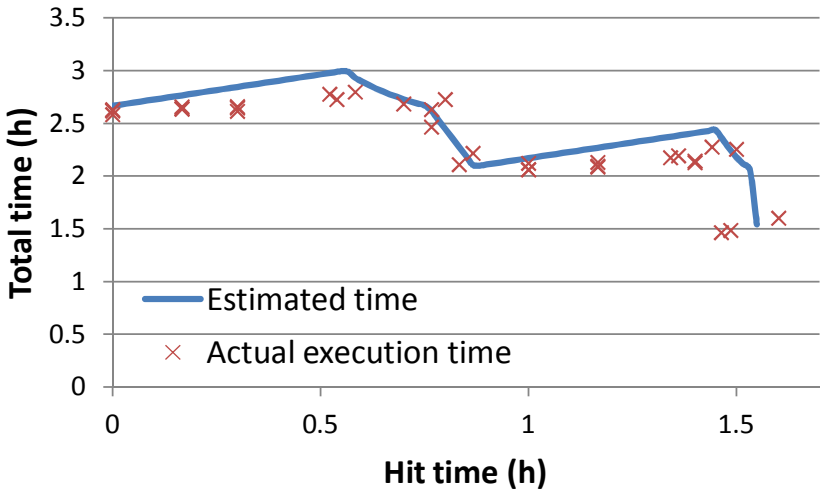

FIGURE 3.14: Estimated vs. actual total execution time of one GNMF iteration as a hit occurs at different times.

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 3.13, we compare the measured execution times with the estimates produced by Cümülön v1, 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).

Figure 3.14 gives the time validation result on one iteration of GNMF instead (recall in Algorithm 3), with a *sync* job after job 3 (according to Cümülön v1's suggestion). As we can see, Cümülön v1's execution time estimates are consistently accurate in both cases.

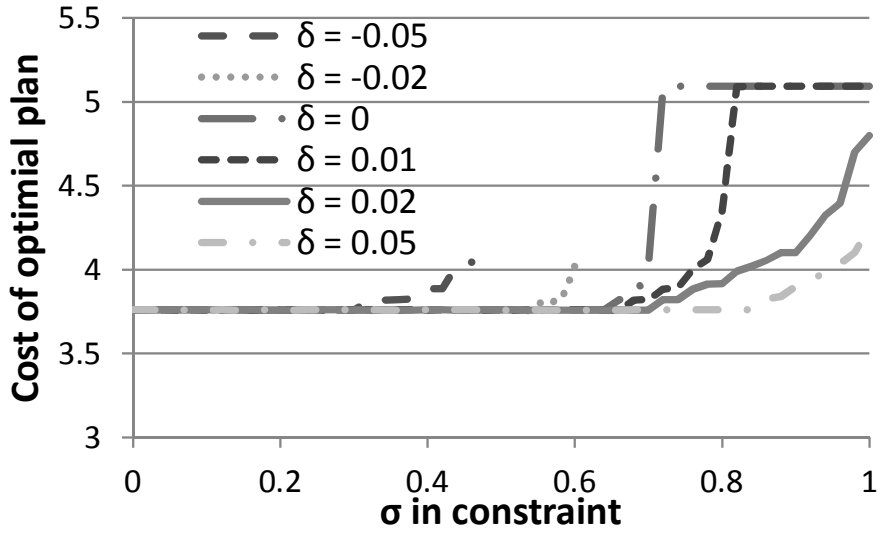

FIGURE 3.15: Estimated expected cost of the optimal plans for RSVD-1 under different constraint settings (values of δ and σ). Three primary nodes used.

3.4.3 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 3.3.5), we simulate 10,000 market price traces using our price model (Section 3.3.1) trained from historical Amazon spot price data in the first six months of 2014 for `c1.medium` in zone `us-east-1a`.

Impact of setting the constraint We first study the impact of the parameters δ and σ in the user-specified constraint on the final optimal plan. Recall the optimization constraint is that the actual cost is within $(1+\delta)$ of the baseline cost with probability no less than σ. Using RSVD-1 with $l = 2$k, $m = n = 100$k, we vary the values of δ and σ and plot the the expected cost of the optimal plans in Figure 3.15.

In general, lower values of δ and higher values of σ lead to a tigher constraint.

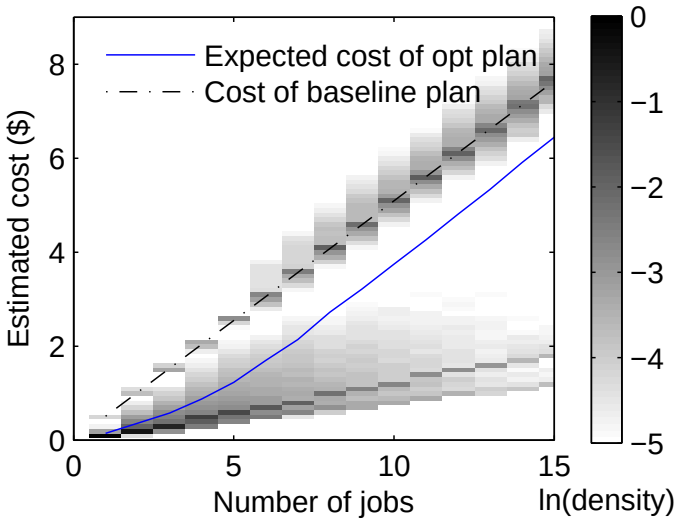

FIGURE 3.16: Estimated cost distributions of optimal plans for RSVD-1 with varying number of jobs, compared with baseline costs.

For each fixed value of δ, when $\sigma = 0$, which means no constraint, the cost of the optimal plan is \$3.76. As σ increases, plans start to be ruled out and the optimal cost starts to increase. For negative values of δ, eventually no valid plan could be found. While for non-negative δ, the baseline plan of not bidding always meets the constraint, thus the optimal cost only downgrades to the baseline cost of \$5.09 for higher σ values. In our default setting where $\delta = 0.05$, even when $\sigma = 1$, meaning that the total cost of the plan should be lower than $\$5.09 \times (1 + 0.05) = \$5.34$ 100% of the time, Cümülön v1 is able to find a feasible plan with expected cost \$4.26, which is still lower than the baseline cost of \$5.09.

Effect of Number of Iterations in RSVD-1 In this experiment, we investigate how longer iterative workloads affect Cümülön v1's optimization decisions. We vary the number of multiplies in RSVD-1 from 1 to 15 (k up to 7). Figure 3.16 shows, for each number of jobs, 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 cost distribution is shown as vertical stripe of PDF in log scale where darker shades indicate higher densities. Additional details about P_{opt} are shown in Table 3.1.

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 v1 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 jobs 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 3.1, we see that as the workflow becomes longer, we tend to bid for more transient nodes at lower prices, up to 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.

It is worth noting the expected amount of cost saving (i.e., the gap between dotted and solid lines in Figure 3.16) converges to around \$1.50 as the number of jobs 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 v1 is allowed to bid again after a hit and when the market price comes down (Section 3.3.9 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 6 jobs

Table 3.1: Bidding and syncing strategies chosen by the optimal plans of RSVD-1 for varying number of jobs in Figure 3.16.

# of jobs	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

(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.

Optimization of GNMF Now let's take a look at how Cümülön v1 optimizes another real program GNMF (Algorithm 3). Recall that it is compiled into six Cümülön v1 jobs per iteration, with job dependencies shown in Figure 3.1. Assume we need to run 10 iterations of GNMF in a $(3, 10)$ c1.medium cluster, with bid price \$0.2. Figure 3.17 shows the estimated cost associated with the optimal sync plan suggested by Cümülön v1, under each possible hit time. Note each small bump in the estimated cost correspond to one of the dominating jobs (e.g. job 1 and job 4 in the first iteration) in the workflow. The upper figure shows the distribution of the hit time collected from simulated price traces. Note that since the bid size is relatively small with regard to the size of the workflow and it takes around 17 hours to finish without a hit. Since we bid relatively high (compared to the on-demand price \$0.145), $10,584$ out of the total $100,000$ price traces result in a longer hit time than 17 hours, meaning there is still a 10% chance that we can finish without a hit.

1) As annotated in Figure 3.17, the best sync plan chosen by Cümülön v1 includes syncing jobs 3, 9, 15, 21, 27 and 33 out of the total 60 jobs. This is because Cümülön v1 models about the distribution of the hit time and decided to sync more during the early stages where the hit is more likely to happen.

2) Cümülön v1 is also able to extract structural information out of the workflow and prioritize the jobs to sync. Note that all the jobs in the optimal sync plan

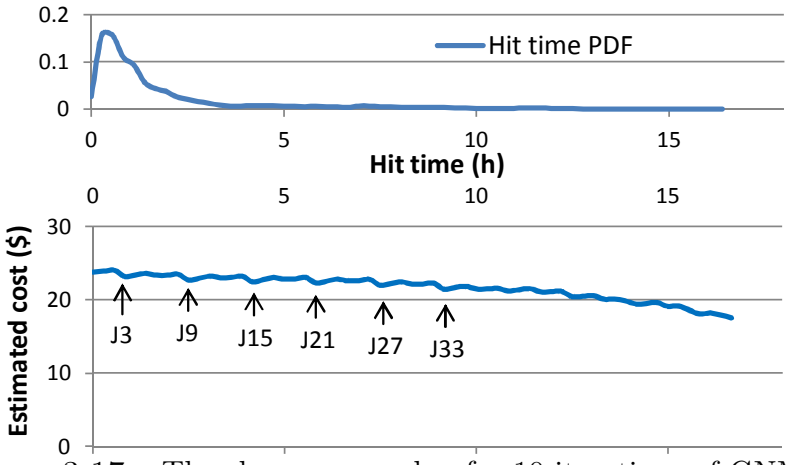

FIGURE 3.17: The chosen sync plan for 10 iterations of GNMF.

corresponds to the same location as job 3 in Figure 3.1. Consider the jobs in the second iteration, job 7 and job 10 are the most time consuming jobs. However, syncing these two jobs is less beneficial because we probably still need to rerun job 6 (job 3) and the jobs before them when hit happens we've lost some results out of job 9 and job 8 (job 12 and job 11). Consequently, the structure of the workflow imply that job 6 and job 3 are more worth syncing. Furthermore, job 6 is less crucial because immediately afterwards, job 7 will be reading job 6's output lots of times, and most of the results will be cached to primary store when during these reads. While the result of job 3 will not be read at all until three more jobs finish. In this case, the Cümülön v1 successfully extracted this structural information and found out that syncing job 3 (and the equivalent jobs in other iterations) is the best choice.

Effect of Bid Price on GNMF Using GNMF, we now examine how the choice of bid price influences cost. In this experiment, we consider two GNMF iterations in a $(3, 10)$ cluster, with three different bid prices: \$0.05, \$0.12, and \$0.25. Cümülön v1 picks the best plan given the bidding strategies.[11] Figure 3.18 plots, for each of the

[11] If we let Cümülön v1 pick the bid price but still limiting to 10 transient nodes, the optimal bid price will be \$0.19, which achieves an expected cost of \$4.37, compared with the baseline cost of \$4.75.

three bid prices, the estimated cost distribution and the average cost over different hit times. The density reflects both the probability of the hit occurring at a given time and the probability of incurring certain cost conditioned on the hit time.

Note that for all three cases in Figure 3.18, 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 v1 places after jobs 3 and 9 (for the same reasons discussed in the previous experiment) 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 a longer duration.

Impact of number of simulated price traces As discussed in Section 3.3.5, simulated price traces are used by the Cümülön v1 Optimizer to evaluate and obtain the cost distribution of every candidate plan. Now using RSVD-1, we investigate how the number of price traces used by the Optimizer influences the quality of the optimal plan found.

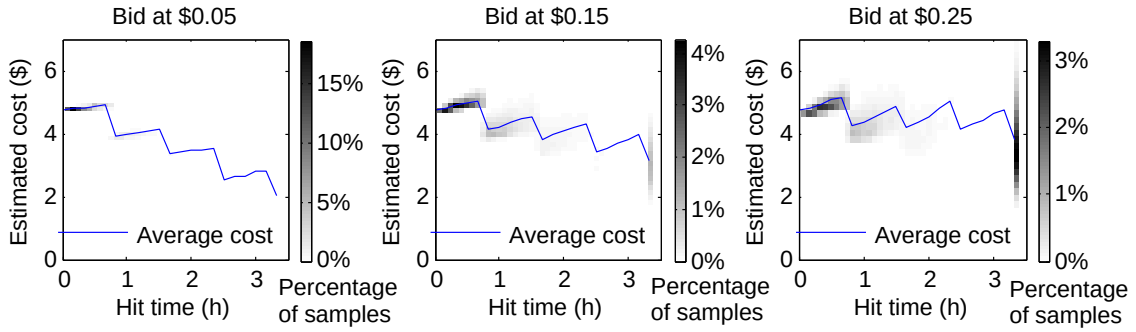

FIGURE 3.18: Effect of bid price on the distribution of hit time and total cost for two GNMF iterations.

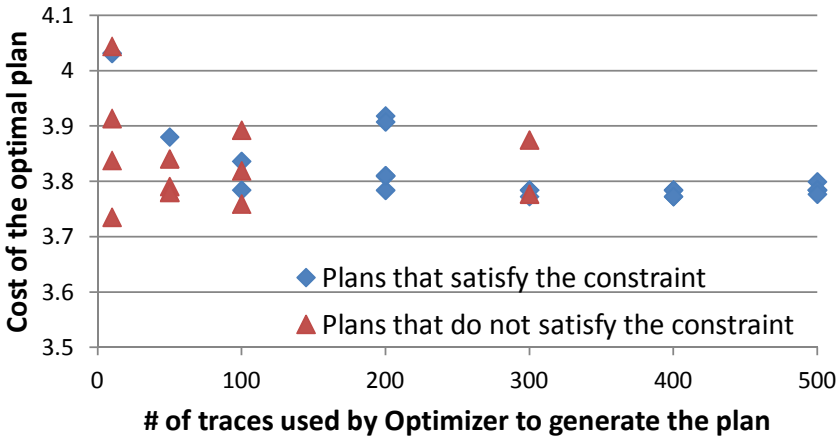

FIGURE 3.19: Quality of the suggested optimal plan of RSVD-1 when the number of simulated price traces used by the Optimizer varies. Five samples are drawn per trace count. $\delta = 0.05$ and $\sigma = 0.9$ are used in the constraint.

Recall that we consistently used a set of $10,000$ simulated price traces in this chapter. In this experiment we treat this whole set of traces as the "ground truth" and use it to evaluate the quality of a plan, i.e. expected cost and whether achieved σ is greater than σ in the constraint. Results are shown in Figure 3.19. For each trace count in x-axis, five sets of price traces are sampled from the full $10,000$ traces. We feed the five trace sets into the Cümülön v1 Optimizer respectively, and get five suggested optimal plans that satisfy the constraint. Lastly, we evaluate the five plans using the $10,000$ traces, plot their expected cost and see whether the plan "really" satisfy the constraint.

As we can see, the cost of the plans has converged well for trace set sizes above 400, and all five plans satisfy the constraint. As the size of the trace set decreases to the left, more plans found by the Optimizer do not "actually" satisfy the constraint. Furthermore, the "actual" expected cost of the plans tend to become larger and have larger variation amongst the five samples. This is because when the number of price traces decreases, the set of traces becomes less and less representative of the true price distribution, and the variance among different sample sets increases. Since the Optimizer treats the trace set as the true distribution, its decision becomes worse.

Although in this case 400 price traces already suffice, we still use $10,000$ traces for all other experiments since our optimization procedure is fast enough and more traces gives us a finer picture of the distributions.

Note on Optimization Time Even though Cümülön v1 derives the cost distribution of each possible plan by repeatedly going through simulated price traces, the total optimization time is reasonable for workloads whose sizes warrant the use of clouds for parallel execution. For example, Figures 1.2, 3.10, 3.20, and 3.22 all require full optimization including the choices of bidding and syncing strategies; in every case, our optimizer completes under 5 minutes on a standard desktop computer with 4-core 3.4GHz Intel i7-2600 CPU and 8GB of memory.

3.4.4 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 v1's flexibility, we can simply plug in this pricing scheme into Eq. (3.1) and see how that it would affect our optimal strategies. Consider the same RSVD-1 workload and recall the plan space shown in Figure 1.2a. We keep all settings the same but change the pricing scheme; the resulting plan space is shown in Fig-

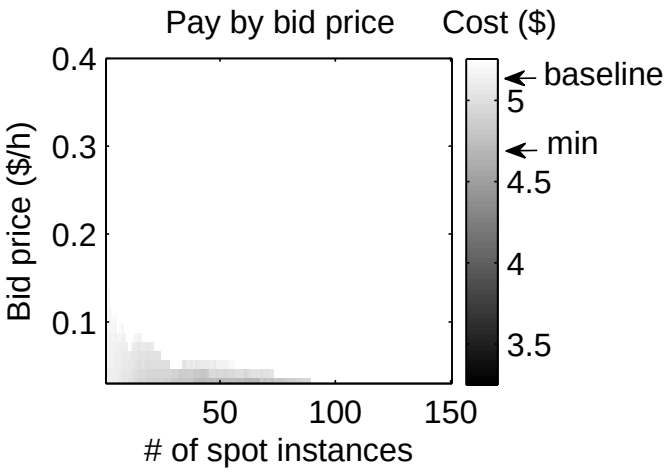

FIGURE 3.20: Estimated expected cost of the optimal plans for RSVD-1 given various bidding strategies, under the pay-by-bid-price scheme.

ure 3.20. 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 1.2a. 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 and the cost of \$3.78 achievable under the pay-by-market-price scheme.

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, Figure 3.21 shows the plan space if all market prices are consistently halved and users are still charged at bid price. As we can see, even with reduced market price, pay-by-bid-price still limits the feasible range of bid prices mostly below \$0.1 (recall the on-demand price of \$0.145). However, by bidding at \$0.02 (assuming p_0 = \$0.01) for 79 spot instances reduces the expected cost to \$4.25 from the baseline cost \$5.09.

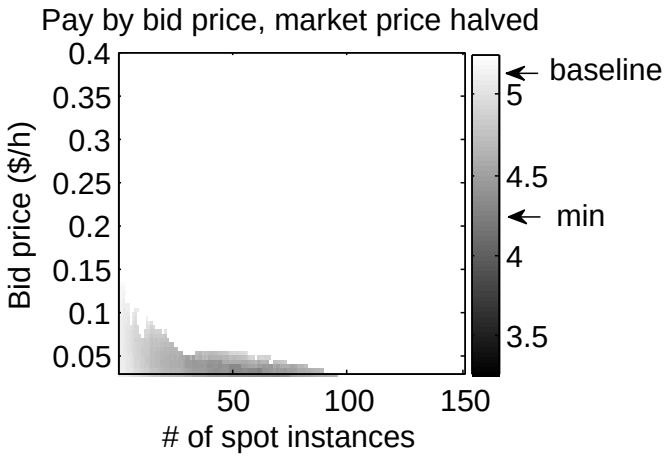

FIGURE 3.21: Estimated expected cost of the optimal plans for RSVD-1 given various bidding strategies, under the pay-by-bid-price scheme, and assuming all market prices are halved.

Amazon's Pricing Scheme As discussed in Section 3.3.2, Amazon EC2 uses a pricing scheme different from what we assume by default. 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 v1 explore the plan space for the same RSVD-1 workload considered in Figures 1.2a and 3.20 under the same settings. The result is shown in Figure 3.22. The feasible plan space now has a jagged boundary because of usage time rounding: the spikes correspond 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 ($3.78).

Since Cümülön v1'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 some work done at a very low cost, or a hit happens and the last partial hour is free anyway. **2)** We

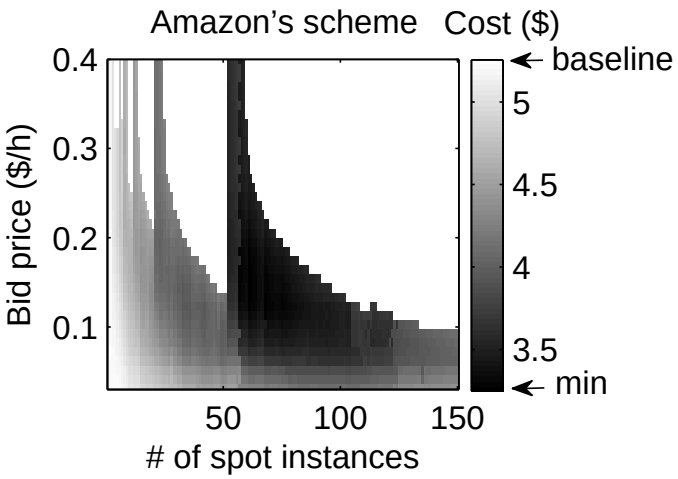

FIGURE 3.22: Estimated expected cost of the optimal plans for RSVD-1 given various bidding strategies, under Amazon's pricing scheme.

will bid again after a hit (Cümülön v1'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 transient nodes 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 v1, i.e., we have 3 primary nodes and always bid for 60 transient nodes.

We compare the cost distributions of Cümülön v1's plan and this "tricky" strategy (thereafter referred to as *tricky*) in Figure 3.23. Overall, *tricky* has an expected cost of $6.86, much higher than Cümülön v1'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 v1 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

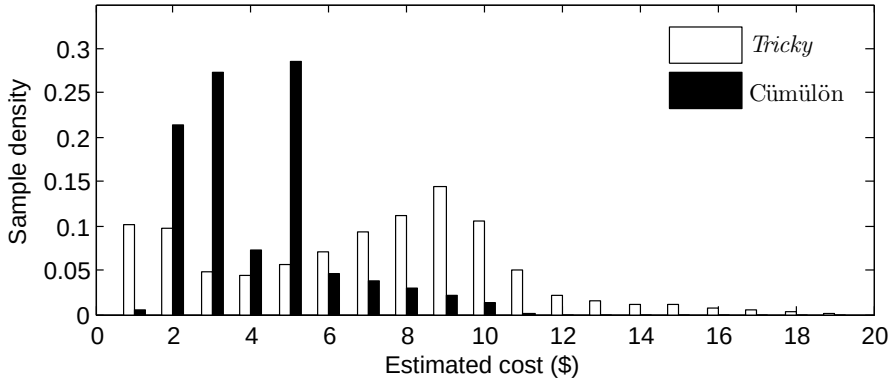

FIGURE 3.23: Comparison of the estimated cost distributions for Cümülön v1's optimal plan and *tricky* for RSVD-1.

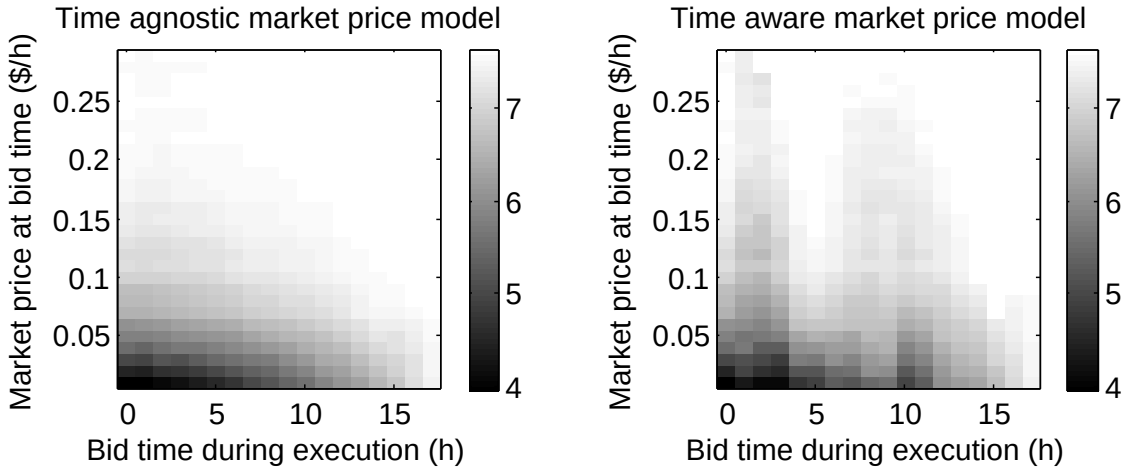

FIGURE 3.24: The cost of the optimal plans under different bid time and market price when bid. RSVD-1 ($k = 7$) with three primary nodes used.

are offset by repeated I/O overhead, 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 v1 optimization despite its various assumptions.

3.4.5 Delayed Bidding

Now we present the results on the *delayed bidding* extension. Recall in Section 3.3.7 that the decision made by Cümülön v1 with *delayed bidding* consists of two parts:

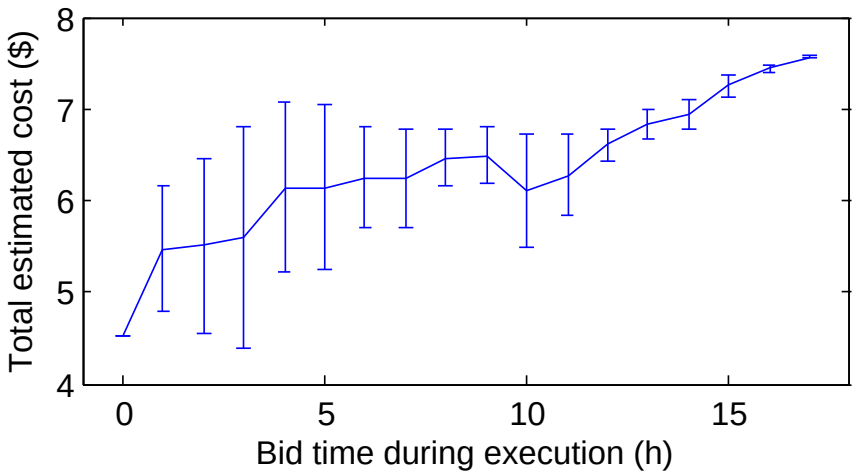

FIGURE 3.25: The mean and standard deviation of the estimated total execution cost of RSVD-1 ($k = 7$) under various bid time, assuming market price at $\mathfrak{T}_0$ is \$0.02.

the bid time $\mathfrak{T}_{\text{bid}}$, and (depending on the market price then p_{bid}) the bidding and syncing strategies to apply at time $\mathfrak{T}_{\text{bid}}$. This decision is made in two steps: first, for all combinations of $\mathfrak{T}_{\text{bid}}$ and p_{bid} , we calculate $\mathfrak{C}(n_{\text{prim}}, \mathfrak{T}_{\text{bid}}, p_{\text{bid}})$, which is the expected cost of the optimal plan that bids at time $\mathfrak{T}_{\text{bid}}$ when the market price then is p_{bid}; secondly, we derive $\mathfrak{C}(n_{\text{prim}}, \mathfrak{T}_{\text{bid}})$ (in Eq. 3.5) by incorporating the distribution of p_{bid}, and then pick the best bid time that minimizes it.

Impact of Bid Time and Price at Bid Time First, let's see how the values from the first step can already help users make informed decisions on the run. Assuming execution starts with three primary nodes at time zero, we visualized the plan space of a slightly long RSVD-1 workflow (with $k = 7$, i.e. 15 multiply jobs) in Figure 3.24. We plot values of $\mathfrak{C}(n_{\text{prim}}, \mathfrak{T}_{\text{bid}}, p_{\text{bid}})$, the expected total cost (Eq. 3.4) of the optimal plan that satisfies the user-specified risk constraint, at given bid time $\mathfrak{T}_{\text{bid}}$ and market price at bid time p_{bid}. This workflow takes 17 hours to finish only wit primary nodes.

The left subplot used the default non-periodic price model described in Section 3.3.1. This price model is agnostic to current time, and only draw samples for next market price and inter-arrival times conditioned on current price. Under this

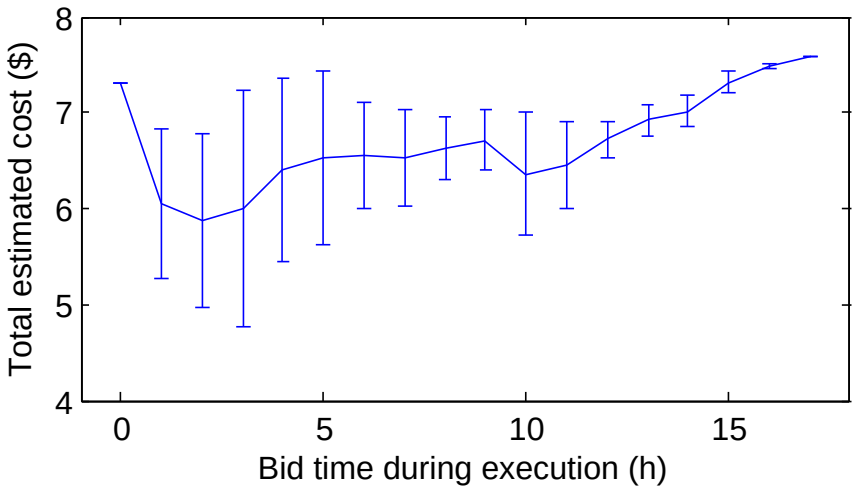

FIGURE 3.26: The mean and standard deviation of the estimated total execution cost of RSVD-1 ($k = 7$) under various bid time, assuming market price at $\mathfrak{T}_0$ is \$0.145.

price model, regardless of when we bid during execution, the higher the market price at bid time, the higher the total cost. This is basically because the spot instances are more expensive in expectation, and the expected duration of the nodes becomes shorter. Less benefit can be obtained by using spot instances. On the other hand, assume that the market price at bid time is fixed, then as we wait more and bid later, the total cost will also increase gradually. This is because primary nodes are doing more and more work, and leaving less work for the cheaper spot instances.

Since in the real price history data, we do observed a correlation between time and the price distribution. The right subplot of Figure 3.24 plots the updated plan space if we used the periodic market price model, which is time-aware because it takes time (time of the day, day of the week) as extra features. In this case, we assume the execution on primary nodes starts at 0am on a Monday. Since in the training data, price tends to stay lower on Monday morning, the Cumulon optimizer found a new region starting around 10am when we can benefit more by bid at that time. It clearlly tells user that if it is alreay around 6am, then it might be better to

wait a little more until the price drops and stay lower after around 10am, and bid at that time.

Overall with this graph, the user can decide to place the bid or wait continously during execution and observing the market price then, depending on how conservative or optimistic she is about future prices.

Decision on the Optimal Bid Time Since the market price model can provide us the expected distribution of future prices, Cümülön v1 can decide the optimal bid time $\mathfrak{T}_{\mathsf{bid}}$ at the very beginning of the execution ($\mathfrak{T}_0$), so that user does not need to keep monitoring the market price and Figure 3.24, trying to decide whether to place to bid, and hoping to have made the right decision.

Once distribution of future market prices (assuming market price at $\mathfrak{T}_0$ is \$0.02) is fed into the right subplot of Figure 3.24, we arrive at Figure 3.25, where we present user with the The mean and standard deviation of the expected total cost (Eq. 3.5) under various $\mathfrak{T}_{\mathsf{bid}}$. Note that the reported standard deviation of costs only comes from the uncertainty of future market prices. For each future bid time, this result incorporates the costs of the optimal plans given all possible market prices at bid time, as well as the modeled probability of realizing each plan. Now user can easily understand and decide when to place the bid and the associated consequence and risks. In this case, the best bid time is at the start of the program ($\mathfrak{T}_0$), beacuse the expected cost is minimized at \$4.52.

When we bid at $\mathfrak{T}_0$, since we know the exact current market price, the cost variance brought by the uncertainty of market price at bid time is zero. On the other hand, the workflow will finish in around 23 hours with no bidding. When the bid time gets closer to the 23rd hour, the variance diminishes to zero essentially because not much work is left when we place the bid. The total cost becomes less and less dependent on p_{bid}. In general, the earlier we bid, the more work can be

Table 3.2: Details on the baseline plans and the optimal plans of RSVD-1 for varying number of primary nodes in Figure 3.27.

n_{prim}	1	3	5	7	9	11	13	15	17	19	21	23	25
baseline time (h)	32.48	11.71	7.21	5.45	4.29	3.59	3.07	2.78	2.43	2.26	2.09	1.92	1.77
baseline cost ($)	4.71	5.09	5.23	5.53	5.60	5.73	5.78	6.06	5.99	6.22	6.36	6.39	6.42
opt exp. cost ($)	4.17	3.78	3.58	3.56	3.49	3.51	3.48	3.50	3.53	3.63	3.72	3.79	3.85
$\hat{p}$	0.71	0.1	0.11	0.1	0.1	0.09	0.08	0.1	0.1	0.1	0.1	0.1	0.1
n_{tran}	2	77	102	102	148	146	144	145	140	138	136	134	145
sync jobs	$\varnothing$	2	3	4,6	5	1,5	6	2,6	2,6	2,6	2,6	2,6	7

done by the cheaper transient nodes, and thus the lower expected cost. At the same time, the earlier we bid, since more work will be done by the transient nodes, the uncertainty of market price at bid time will have a stronger impact on the variance of the total cost. However in this example, we do see a case that if for some reason the user decides not to bid within the next three hours, then it is actually better to wait until the 10th hour and then bid, because the market price model suggests that the price is likely to be lower and stay lower for a longer period then.

Now if the current market price observed at $\mathfrak{T}_0$ is \$0.145 rather than \$0.02, Figure 3.25 becomes Figure 3.26, with all other settings unchanged. Since the price at the start of the program is much higher, now Cümülön v1 suggests to wait for two hours and then place the bid. Note that comparing to Figure 3.25, the estimated total cost of bidding in the first four hours shifted higher. This because the market price model captures the higher current market price, and believes that in the near future, the overall price will be higher than in the case of Figure 3.25. On the other hand, however, as the bid time becomes larger (beyond five hours), the trend becomes closer to Figure 3.25, because the known market price difference at time zero has less and less impact on the expected difference on future prices.

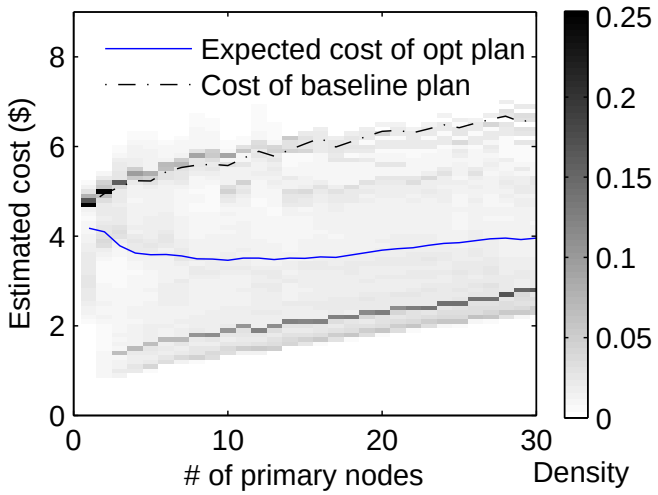

FIGURE 3.27: Estimated cost distributions of the optimal plans for RSVD-1 under varying baseline plans (primary cluster sizes).

3.4.6 Optimizing the Primary Size

Now we show the results on including the primary size into the optimization space as discussed in Section 3.3.8. Figure 3.27 shows the cost distributions of the optimal plans of RSVD-1 under various primary cluster sizes. The dotted line is the estimated cost of the baseline plan of not bidding. It increases with primary size because of the scaling overhead. The solid line is the expected cost of the augmented optimal plans, whose cost distribution is shown in the vertical strips of probability densities.

As we can see, the cost distributions gradually shift up as primary size increases. This is because the primary nodes are more expensive than transient nodes, using more of them will in general increase total cost. However, we do see a decreasing expected cost of the augmented plans when primary size are below ten. This benefit is brought by the first impact discussed in Section 3.3.8: higher primary store bandwidth. With more primary nodes, the data egress and ingress from primary store are much faster. As a result, more transient nodes can be utilized with higher efficiency. Overall, the minimal expected cost is achieved at $3.46 with ten primary

nodes. More details on the plans in Figure 3.27 is shown in Table 3.2.

For RSVD-1, if the user-specified deadline is twelve hours, then the optimal baseline plan has three primary nodes. The optimal augmented plan based on $n_{\text{prim}} = 3$ will have an expected cost of \$3.78. However, if we increase the primary cluster to ten nodes. The optimal expected cost can be lowered to \$3.46 even though the new baseline plan itself has a higher cost. On the other hand, if the user specifies a deadline of two hours, then best baseline plan is $n_{\text{prim}} = 23$. However, Cümülön v1 will not consider n_{prim} smaller than 23 because these baseline plans (and thus the augmented plans) cannot meet the deadline for sure.

3.5 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* [16] uses high degrees of redundancy to achieve reliability; *Spot Cloud MapReduce* [32] depends on reliable external storage services rather than local storage in the cluster. Both methods have negative performance implications. Like Cümülön v1, a number of systems use more reliable primary nodes for storage. Chohan et al. [9] 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* [2] 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 3.2.1) and verified (Section 3.4.1), 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 works [39, 44, 46, 3, 38, 26, 47] studied checkpointing and bidding strategies under various settings in order to satisfy service level agreements, meet deadlines, or minimize cost. Others [33, 37] 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* [48] are able to infer which units of computation to rerun in order to recover from failures. As discussed in Section 3.2, thanks to declarative program specification, Cümülön v1 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 v1 aims at helping users of a public cloud, others [49, 45] 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 3.4.4).

3.6 Conclusion

In this chapter we have presented Cümülön v1, 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 v1'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 v1 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 v1 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 v1 approach will be an interesting direction to further explore.

4

Cümülön v2

4.1 Introduction

As mentioned in Chapter 1, the auction-based computing resources (e.g. Amazon EC2 spot instances) raised an interesting option for cloud users. On one hand, such resources are usually much cheaper than the reliable fixed-price on-demand instances; on the other hand, the problem of sudden massive uncorrelated reclamation of spot nodes poses a great challenge on building Cümülön: how do we ensure data integrity and progress with minimal system overhead, and furthermore, how/when to bid for the spot nodes?

Cümülön v1 (described in Chapter 3) is our first approach solving the problem. It comes with a smart and efficient system design. However, system complexity complicates cost estimation, which consequently limits the optimizer to one single batch of spot instances. Exploring the trade-off between the system complexity and optimizability, we now consider an alternative design with an easier-to-model system that allows a smarter optimizer that can dynamically bid for multiple batches of spot instances.

In this chapter, we present this alternative system which we call Cümülön v2. Assuming there is an external scalable storage system that can store all data, we no longer need to worry about data loss any more. Cümülön v2 simply maintains a dynamic pool of spot instances for execution. It adjusts the number of nodes to use according to the observed market price then, in order to minimize the total execution cost under acceptable cost variance, given a user-specified deadline. The problem space (e.g. time) is continous, so we discretize it and model the problem as a Markov Decision Process (MDP) and solve the problem offline. The model's solution is a policy that instructs the optimal action (cluster size) to take in every possible future state. Next, we discussed how to apply the discretized policy during the continous real execution. Simulation is used to experimentally study the performance of proposed policies.

4.2 Model Setup

We assume that the cloud provider will only announce the current market price p, without any information on future prices. Price changes can be announced at any time. Users acquire machine by placing bids in the form of $(\hat{p}, n)$, where $\hat{p}$ is the bid price and n is the bid size. Once a bid is placed, it cannot be modified. Users are charged in units of T_{charge} defined by the cloud provider. Specifically, usage time is rounded up to a multiple of T_{charge} and each usage duration of T_{charge} gets charged by the market price at the beginning of the unit. For instances, Amazon uses an hour (i.e. $T_{charge} = 3600s$) and additionally does not charge the last partial hour if the nodes are reclaimed. While the Google cloud charges user by minutes ($T_{charge} = 60s$).

Assume we have J jobs to execute sequentially. Each job consists of a configurable number of parallel tasks. Assume we only use machines of identical settings rent from a single spot market. [1] Let $T_j(n)$ denote the execution time of job j ($j = 1, ..., J$)

[1] The case of multiple markets are left to future work.

using n nodes, then $w_j = T_j(1)$ represents the total machine time required to finish job j, or in other words, the total amount of work to do in job j. Define $W = \sum_{j=1}^{J} w_j$ to be the total amount of work in the workflow.

Define $g_j(n) = \frac{T_j(1)}{T_j(n)}$ to be the speedup function (well studied in [13]) of job j. Because of the increasing parallelism overhead when n increases, $g(n)$ should be an increasing and concave function satisfying $g_j(n) \leqslant n$ and $g_j(1) = 1$. (Note that our model does not need any of these assumptions on $g_j(n)$.) We have $w_j = T_j(n)g_j(n)$. This shows that $g_j(n)$ is also the average amount of work accomplished in one unit time during job j, using n machines.

Whenever the number of machines in the cluster is changed, overhead will be incurred. When increasing n, the new nodes will need some time for system initialization, without doing actual work. When decreasing n, running tasks are killed and thus need to be rerun later, but we still need to pay for the wasted machine time. Since we assume that all tasks write results to both the local store as well as a remote reliable store, finished tasks/jobs will not be affected when nodes are gone. In general, define $o(n_1, n_2, w) \geqslant 0$ to be the overhead, or the equivalent amount of work lost, when we change the cluster size from n_1 to n_2 and there is w amount of work left in the workflow then.

4.3 Optimization Problem

In general, given a deadline D, we are looking for a provisioning plan that encodes the n to use at every time point before D, such that the total monetary cost is minimized while ensuring the workflow can finish by the deadline. Picking an n every second is computationally expensive, and is also an overkill since any realistic plan will not keep changing n. In the model, we introduce an optimization time unit T_{opt} and only pick one n_t for each time step t (a duration of length T_{opt}).

We discretize possible spot instance market prices into a price set S_p with N_p distinct values. Let p_t denote the market price at time $t \times T_{opt}$ (beginning of time step t). Again we assume that the market price may change at any time.

Before we introduce uncertainty and probabilities (in Section 4.5), the optimization problem can be formalized as follows: given a deadline D, pick n_t ($t = 0, ..., D/T_{opt} - 1$) to minimize $Cost(\{n_t\})$ the total monetary cost charged, subject to $\sum_{t=0}^{D/T_{opt}-1}[g(n_t)T_{opt} - o(n_{t-1}, n_t, w)] \geqslant W$, assuming $n_{-1} = 0$.

In general the cost term $Cost(\{n_t\})$ is hard to write down analytically because of the discrepancy between T_{opt} and T_{charge}. T_{charge} is defined by the cloud provider, but we can choose T_{opt} ourselves. A smaller T_{opt} tends to give us more fine-grained and better quality result, but at the same time means longer optimization time. In this work, we limit the choice of T_{opt} to two cases: either $T_{charge}|T_{opt}$ or $T_{opt}|T_{charge}$. Here | means divides.

- If $T_{charge}|T_{opt}$, each machine in n_t will be charged T_{opt}/T_{charge} times during time step t. We have $Cost(\{n_t\}) = \sum_{t=0}^{D/T_{opt}-1} n_t \sum_{i=0}^{T_{opt}/T_{charge}-1} p_{t+i(T_{charge}/T_{opt})}$. When $T_{opt} = T_{charge}$, $Cost(\{n_t\})$ simplifies to $\sum_{t=0}^{D/T_{opt}-1} n_t p_t$.

- If $T_{opt}|T_{charge}$, let $k = T_{charge}/T_{opt}$, then at the beginning of time step t, not all n_t machines will be charged, because some of them is already charged at earlier time steps. Let m_t denote the number of machines that actually get charged at time step t.

  If we assume reclamation would not happen[2] and that once a node is charged for another T_{charge} we do not terminate it within the duration, then we have $n_t \geqslant \sum_{i=1}^{k-1} m_{t-i}$. This means that all the nodes we have paid for in the previous

[2] For now, assume a bid price of infinity so that no reclamation would happen. Of course this is not necessarily a practical strategy. Later we will discuss how Cümülön v2 set the actual bid price when carrying out the plan.

$k-1$ time steps will be running in the current time step for sure. If we decided to pay for m_t nodes now (could be running ones renewed from m_{t-k} or new ones if $m_t > m_{t-k}$), we have $n_t = \sum_{i=0}^{k-1} m_{t-i}$ for all t. Reversely, m_t can also be derived from n_t:

$$m_t = \begin{cases} 0 & \text{if } t < 0 \\ n_t - \sum_{i=1}^{k-1} m_{t-i} & \text{if } t \geqslant 0 \end{cases}$$

Overall, we have $m_t, n_t \geqslant 0$ and $Cost(\{n_t\}) = \sum_{t=0}^{D/T_{opt}-1} m_t p_t$.

Assuming either $T_{charge}|T_{opt}$ or $T_{opt}|T_{charge}$ is true, if we define $k = \lceil T_{charge}/T_{opt} \rceil$, then when $T_{charge}|T_{opt}$, we have $k = 1$ and $m_t = n_t$. We can unify the two cases into one:

$$Cost(\{n_t\}) = \sum_{t=0}^{D/T_{opt}-1} m_t Charge(t) \tag{4.1}$$

Where $Charge(t) = \sum_{i=0}^{\lceil T_{opt}/T_{charge} \rceil - 1} p_{t+i(T_{charge}/T_{opt})}$ is the total charges incurred on each machine during time step t.

4.4 Deterministic Future Price

Solving the problem is straightforward when the future market price trace is exactly known. There are two cases.

When $T_{charge}|T_{opt}$, and if the cluster changing overhead is assumed trivial, i.e. $o(n_1, n_2, w) = 0$, the problem can be solved with a greedy algorithm, keep allocating the work to the most price/work efficient time step. Specifically, starting from $n_t = 0$ for all t, we always pick the t that maximizes $(g(n_t + 1) - g(n_t))/Charge(t)$ and increment n_t by one. Here $Charge(t)$ (defined in Equation 4.1) is constant for every t since we know the future price trace. Simply keep doing this until $\sum_{t=0}^{D/T_{opt}-1} g(n_t) \geqslant W$ is satisfied. A formal proof is given in Appendix A.

Otherwise, the choices of n_t is interrelated and greedy no longer works. Instead, we can solve this problem with dynamic programming. Let $C(t, w, [m_{t-1} \cdots m_{t-k}])$ denote the minimal cost of finishing w amount of work left starting from time step t, and that $m_{t-1} \cdots m_{t-k}$ machines are charged in the beginning of the previous k time steps. We have the state transition function:

$$C(t, w, [m_{t-1} \cdots m_{t-k}]) = \min_{m_t > 0} \{m_t Charge(t) + C(t+1, w', [m_t \cdots m_{t-k+1}])\}$$

Where $n_t = \sum_{i=t-k+1}^{t} m_i$ and $n_{t-1} = \sum_{i=t-k}^{t-1} m_i$. $w' = w + o(n_{t-1}, n_t, w) - g(n_t)T_{opt}$ is the amount of work left for the next time step. For implementation, additional tweaks including discretizing the w dimension is needed. The time complexity of the DP is $O(\frac{D}{T_{opt}} W N^{k+1})$, where N is the maximum number of nodes to use simultaneously. Because of the parallelism overhead (speedup function $g(n)$ converges to a constant for n large enough), $\{n_t\}$ are naturally bounded.

Later in the experiments, we will use this DP algorithm as the "oracle" that gives us the minimal achievable cost for each price trace from the test set. It serves as a theoretic lower bound for our probablistic optimizer who does not know the exact future.

4.5 Probabilistic Future Price: An MDP Model

Under the setting where future market price is unknown beforehand, we follow the intuition to dynamically adjust n_t during execution according to p_t (market price of the nodes then) as well as the amount of work and time left before the deadline.

A probabilistic model for future price evolvement can be trained using historical price data. With this model, we can estimate the values we are interested in, for instance, $Charge(t)|p_t$ the expected charge function given the market price then. We can also derive a price transition function from the trained price model. Let $\mathbb{P}^{u\,v}_{t,\Delta t}$

denote the probability that the price becomes v at time $(t + \Delta t) \times T_{opt}$ given that it was u at time $t \times T_{opt}$, i.e. $\mathbb{P}^{u\,v}_{t,\Delta t} = \mathbb{P}(p_{t+\Delta t} = v | p_t = u)$.

Now we can build an MDP model where state $\{t, w, u, [m_{t-1} \cdots m_{t-k}]\}$ represent the situation that when we arrive at the beginning of time step t with w amount of work left to finish before deadline D; the market price then is u; $[m_{t-1} \cdots m_{t-k}]$ machines are charged in the previous k time steps and thus $n_{t-1} = \sum_{i=t-k}^{t-1} m_i$ machines are running before we choose m_t or n_t for time step t. At this state, if we take the action to pay for $m_t \geqslant 0$ machines and thus use $n_t = \sum_{i=t-k+1}^{t} m_i$ machines for time step t, we will incur an immediate cost of $m_t Charge(t)$, and then transition to a certain subset of states (dependent on our choice m_t) at time step $(t + 1)$ with certain probability distribution.

Define policy $\mathcal{P} : \{t, w, u, [m_{t-1} \cdots m_{t-k}]\} \to m_t$, which maps any given state to an action m_t. Let $\mathcal{C}(\mathcal{P}, \{t, w, u, [m_{t-1} \cdots m_{t-k}]\})$ denote the expected cost to finish the rest of the work from state $\{t, w, u, [m_{t-1} \cdots m_{t-k}]\}$ following policy $\mathcal{P}$, then:

$$\mathcal{C}(\mathcal{P}, \{t, w, u, [m_{t-1} \cdots m_{t-k}]\}) = E\left[Cost\left(\mathcal{P}, m_t = \mathcal{P}(t, w, u, [m_{t-1} \cdots m_{t-k}])\right)\right]$$

Where $Cost(\mathcal{P}, m_t)$ denote the cost if we choose to pay for m_t machines for time step t, and then follow policy $\mathcal{P}$. Note that $Cost(\mathcal{P}, m_t)$ is a random variable, with expectation:

$$E(Cost(\mathcal{P}, m_t)) = m_t Charge(t) + \sum_{v \in S_p} \mathbb{P}^{u\,v}_{t,1} \mathcal{C}\left(\mathcal{P}, \{t + 1, w', v, [m_t \cdots m_{t-k+1}]\}\right)$$

Where $w' = w + o(n_{t-1}, n_t, w) - g(n_t) T_{opt}$. Overall, this is an MDP of finite time space, and the constraint is that at time step D/T_{opt}, we must be in a subset of states with $w = 0$. Note that here the state space covers all possible (discretized) futures, and the policy encodes the best action n_t (or m_t) to take at all these future states.

For instance, let $\mathcal{P}_{min}$ denote the policy that always minimizes expected cost of

the plan without looking at variance at all, then:

$$\mathcal{P}_{min}(t, w, u, [m_{t-1} \cdots m_{t-k}])) = \arg\min_{m_t \geqslant 0} E(Cost(\mathcal{P}_{min}, m_t))$$

Risk and variance Besides minimizing the expected cost, we also want to control the risk in terms of the cost. Let

$$\mathcal{V}(\mathcal{P}, \{t, w, u, [m_{t-1} \cdots m_{t-k}]\}) = Var\left[Cost\left(\mathcal{P}, m_t = \mathcal{P}(t, w, u, [m_{t-1} \cdots m_{t-k}])\right)\right]$$

denote the cost variance if we finish the work from state $\{t, w, u, [m_{t-1} \cdots m_{t-k}]\}$ following policy $\mathcal{P}$. We can expend the variance term of $Cost(\mathcal{P}, m_t)$ using the law of total variance as follows:

$$\begin{aligned}
&Var(Cost(\mathcal{P}, m_t)) \\
&= E_{p_{t+1}}\left(Var(Cost(\mathcal{P}, m_t)|p_{t+1})\right) + Var_{p_{t+1}}\left(E(Cost(\mathcal{P}, m_t)|p_{t+1})\right) \\
&= E_{p_{t+1}}\left(Var(Cost(\mathcal{P}, m_t)|p_{t+1})\right) + E_{p_{t+1}}\left(\left[E(Cost(\mathcal{P}, m_t)|p_{t+1})\right]^2\right) \\
&\quad - \left\{E_{p_{t+1}}\left(E(Cost(\mathcal{P}, m_t)|p_{t+1})\right)\right\}^2 \\
&= E_{p_{t+1}}\left(Var(Cost(\mathcal{P}, m_t)|p_{t+1}) + \left[E(Cost(\mathcal{P}, m_t)|p_{t+1})\right]^2\right) \\
&\quad - \left\{E_{p_{t+1}}\left(E(Cost(\mathcal{P}, m_t)|p_{t+1})\right)\right\}^2 \\
&= \sum_{v \in S_p} \mathbb{P}_{t,1}^{u\,v}\left\{\mathcal{V}(\mathcal{P}, ...) + \left[\mathcal{C}(\mathcal{P}, ...)\right]^2\right\} - \left[\sum_{v \in S_p} \mathbb{P}_{t,1}^{u\,v} \mathcal{C}(\mathcal{P}, ...)\right]^2
\end{aligned}$$

Where ... in $\mathcal{V}(\mathcal{P}, ...)$ and $\mathcal{C}(\mathcal{P}, ...)$ above stands for $\{t+1, w', v, [m_t \cdots m_{t-k+1}]\}$.

Cümülön v2's Policy $\mathcal{P}_r$ From users' perspective, the risk brought by the uncertainty of future market prices is a serious concern , Cümülön v2 adopts a family of policies that considers the cost variances as well as the expected cost, specifically:

$$\mathcal{P}_r(t, w, u, [m_{t-1} \cdots m_{t-k}])) = \arg\min_{m_t \geqslant 0} \left(E(Cost(\mathcal{P}_r, m_t)) + r\sqrt{Var(Cost(\mathcal{P}_r, m_t))}\right)$$

where r is a risk trade-off parameter specified by user. Note that when $r = 0$, the policy becomes $\mathcal{P}_{min}$ as discussed earlier.

Solving the MDP Given a policy, we can solve the Markov chain with DP, starting from $t = D/T_{opt} - 1$ and computing the cost expectation $\mathcal{C}(...)$ and variance $\mathcal{V}(...)$ of every state backwards. The computational complexity is $O(DWN^{k+1}N_p^2)$, where N is the maximum number of nodes to use simultaneously, N_p is the discretized possible number of market prices.

We can push the complexity down to $O(DWN^kN_p(N + N_p))$ with preprocessing. Specifically, for each combination of $\{t, u, [m_t \cdots m_{t-k+1}]\}$, we pre-calculate $\sum_{v \in S_p} \mathbb{P}_{t,1}^{u\,v} \mathcal{C}\left(\mathcal{P}, \{t+1, w', v, [m_t \cdots m_{t-k+1}]\}\right)$ for all possible w', which takes time $O(DWN^kN_p^2)$. Next for each state $\{t, w, u, [m_{t-1} \cdots m_{t-k}]\}$, we iterate through all possible m_t and use the best one, which incurs $O(DWN^{k+1}N_p)$.

Currently the discretization along each dimension (time, work, price) is done in a equal-stepped way. More advanced scheme, e.g. prioritizing on the time dimension, can also be employed.

Simplified state assuming no cluster change overhead When $T_{charge} | T_{opt}$, we have $k = 1$ and $m_t = n_t$. If cluster switching overhead is assumed to be zero, i.e. $o(n_1, n_2, w) = 0$, then the choices of n_t become independent. We can remove $[m_{t-1} \cdots m_{t-k}]$ out of the state, and arrive at a simplified state transition function:

$$C(t, w, u) = \min_{n_t \geqslant 0} \left\{ n_t Charge(t) + \sum_{v \in S_p} \mathbb{P}_{t,1}^{u\,v} C(t+1, w - g(n_t)T_{opt}, v) \right\}$$

The computation complexity is now $O(DWNN_p^2)$. With the same pre-compute trick, it reduces to $O(DWN_p(N + N_p))$.

Market signal delay and the skyline bidding scheme Now we start talking about setting the bid price.

Intuitively, with policy $\mathcal{P}$ at hand, we could set a maximum bid price as assumed earlier in the model (Footnote 2), so that we can always hold the nodes as long as the model thinks they are worth the current price u (i.e. $m_t = \mathcal{P}(t, w, u, [m_{t-1} \cdots m_{t-k}]) > 0$) and do not have to worry about reclamation. Once the market price u becomes unacceptably high (i.e. $m_t = \mathcal{P}(t, w, u, [m_{t-1} \cdots m_{t-k}]) = 0$), we simply terminate our nodes before they will be charged for the next T_{charge} amount of usage time.

This idea is also employed in [29]. However, they neglected the potential problem of bidding at maximum price. Depending on the market setup, there could be an unfortunate situation where the market price surges from p_{low} to p_{high} (very high) right before the nodes are charged for the next T_{charge} and after the price change we cannot terminate the nodes in time to avoid the high charge.

In general, these are tradeoffs on setting the bid price. Firstly, we don't want the bid price to be too low, so that we will not suddenly lose nodes that we want to keep. On the other hand, we do not want to set the bid price too high, because it is a safeguard against sudden price surge.

We noticed that whether the unfortunate situation above could happen depends on the **market signal delay** $(\tilde{t_1}, \tilde{t_2})$ defined as follows: if we send out the signal to bid and start a node at time t_1 and later send out the signal to terminate node at time t_2, then in a market with signal delay $(\tilde{t_1}, \tilde{t_2})$, the node is actually charged for usage duration $t_1 + \tilde{t_1}$ to $t_2 + \tilde{t_2}$, in other words, $\sum_{i=0}^{\lceil (t_2+\tilde{t_2}-t_1-\tilde{t_1})/T_{charge} \rceil - 1} p_{(t_1+\tilde{t_1}+iT_{charge})/T_{opt}}$. [3]

- If the market is setup such that $\tilde{t_1} = \tilde{t_2} = 0$, then assuming users can make decisions and act instantly upon price change, the unfortunate case can be avoided for sure. Bidding at maximum prices is risk-free and the best choice because it is the best bidding strategy that sticks to Cümülön v2's optimal policy.

[3] In Amazon EC2, $\tilde{t_1}$ is around five minutes, since it includes the time it takes to bootstrap the new instance; while $\tilde{t_2}$ is around several seconds.

- Otherwise, the unfortunate case could happen. To guard against it we should bid at the maximum price we are willing to pay for the nodes rather than the maximum possible price. We can easily retrieve this information from Cümülön v2's policy $\mathcal{P}$. Specifically, at time step t with w amount of work left, we should bid at $p_{max}(t, w, \mathcal{P}) = \min\{u \mid n_t = m_t = \mathcal{P}(t, w, u, [0 \cdots 0]) = 0\}$. Here we set $m_{t-1} \cdots m_{t-k}$ to all zeros so that we only look at the decisions assuming no node is already running. This is essentially the "skyline" of policy $\mathcal{P}$. Note that bidding at $p_{max}(t, w, \mathcal{P})$ is not a significant deviation of the policy itself because whenever the market price go above it, the policy will inform us to stop paying for more new nodes anyways.

  In general, the closer to the deadline (larger t), or the more work left (larger w), the higher the skyline $p_{max}(t, w, \mathcal{P})$ tends to be. In the assumed market where bid price cannot be changed, the nodes that started earlier will likely have a lower bid price than the desired bid price later, leading to potentially undesired reclamations. As a result, we introduce a constant c_t and bid at $\hat{p}(t, w, \mathcal{P}) = \min_{t \leqslant x \leqslant t+c_t} p_{max}(x, w, \mathcal{P})$ instead. The constant c_t presents a tradeoff between potential higher price charged on nodes (bigger c_t and thus higher bid price) and potential undesired reclamation (smaller c_t and thus lower bid price).

4.6 Applying the Policy in Execution

In this section, we will discuss how Cümülön v2 dynamically executes model suggested policy $\mathcal{P}$ during real execution. Recall that in optimization, we need to discretize each dimension in the plan space. Furthermore, since the optimization grows exponentially with $k = \lceil T_{charge}/T_{opt} \rceil$, we might not afford a small T_{opt}. As a result, the policy offered by the optimizer contains only suggested n to use on discrete points in the actual continuous state space. In the time dimention, whenever a change in

the market price is announced, it is often better to act (adjust n) instantly rather than wait until the next decision points in the plan.

Observing this fact, our simulator is implemented with two **execution modes**: the **strict mode** and the **dynamic mode** (default). In strict mode, we strictly stick to the policy, only consult the policy and change cluster size n at time points tT_{opt} where t is an integer. Specifically, Cümülön v2 will evaluate the amount of work left, the current time and market price, and then consult the policy (by looking at the proper states nearby and combine suggestions if necessary) and then act on changing n if suggested.

While in dynamic mode, the plan from the optimizer are considered as a guideline or "cookbook". Cümülön v2 will consult the cookbook and update n under one of the three conditions: 1. there is a change in the current market price; 2. some running nodes about to be charged for another T_{charge}; 3. The time since the last check exceeds some predefined threshold. Note the third case is to account for runtime variation and deviation from estimations so that we can adjust course accordingly.

Cümülön v2 enforces a rule of **delayed termination** during execution, which means that once we paid for a node for the next T_{opt} time, we will not terminate it ourselves within the duration. (Note that it could still be reclaimed because of price surge.) Since this rule is implicitly built-in to the optimizer, it is automatically enforced in the strict mode.

However in the dynamic mode, delayed termination is cruicial because potentially we make adjustments to n more frequently than what the policy suggests, and we should not waste any machine time we have already paid for. Imagine the case where market price first drops and we are advised to increase n. Shortly after the new nodes are up and charged, price surges to a very high value. Another update will be triggered and the cookbook might suggest decreasing n. Since we have already paid for the new nodes for another T_{charge}, we should not terminate them immediately as

suggested. Rather, we should keep the charged nodes for execution until they reach their next charging point. Then we will consult the cookbook again (according to condition 2 above), and decide whether to really terminate the nodes or extend the lease, depending on the market price then.

Execution simulations It is impractical to run the model proposed policies in real spot cluster for many times to arrive at a statistically sufficient performance evaluation. We use execution simulation instead. The simulator is implemented at the level of tasks (of jobs in the workflow) and slots (of nodes in the cluster). Jobs are executed sequentially. Tasks in the running job are scheduled to available slots in the cluster, just like in a real execution. Upon scheduled, the execution time of the task is randomly drawn from a normal distribution with the mean being the estimated task time from the cost model and a pre-configured variance. When new nodes join the cluster, a preconfigured time is assumed for bootstrapping before its slots becomes available for tasks execution. Whenever we decide to shut down a node, all its running tasks fail and will be rescheduled later. Again, since we assume that all tasks write results directly to a reliable storage, finished tasks will not be affected when shutting nodes down.

Given a specific price trace and a policy, the simulator can carry out the specified plan against the market price trace and report the total running time, cost and all the decisions made during the execution process. Some key features of the simulator include support for bid price (specific for individual node), various charging scheme with different T_{charge}, as well as market signal delay $(\tilde{t_1}, \tilde{t_2})$.

4.7 Experiments

The experiments are conducted in the context of Amazon EC2, although we do not limit the charging scheme of spot instances to Amazon's scheme. **RSVD-1**

($\mathbf{G} \times (\mathbf{A} \times \mathbf{A}^\intercal)^k \times \mathbf{A}$, with $l = 2048$, $m = n = 102,400$, $k = 5$) and **GNMF** (with $k = 100$ and a 7510016×640137 word-doc matrix $\mathbf{V}$ derived from a 2.5GB wiki text corpus) is used as the workload.

Estimation models for the matrix workloads are trained beforehand with run time data collected from benchmark runs. By default *c1.medium* instance type is used.

A market price model is developed (please refer to Appendix B for details), and trained with Amazon's historical spot instance prices, specifically prices of *c1.medium* from zone *us-east-1a*, which is one of the most challenging markets with lots of spikes and high-price regions. We used the model to generate a training set and some test sets of simulated market price traces. The training set is used to estimate the future price transition matrix $\mathbb{P}_{t,1}^{u\,v}$ used in the MDP model, as well as the charging function $Charge(t)$ in Equation 4.1 when $T_{opt} > T_{charge}$.[4] The test set of simulated traces is used to evaluate the plan (policy) under simulated execution discussed in Section 4.6. Two test sets $S_{0.02}$ (default) and $S_{0.2}$, each consisting of $100,000$ price traces, are used in the experiments. The starting prices (p_0) of the traces in $S_{0.02}$ and $S_{0.2}$ are $\$0.02/h$ (the most common seen market price) and $\$0.2/h$ respectively. Note that the on-demand price for *c1.medium* instances is $\$0.145/h$.

By default Cümülön v2 uses policy $\mathcal{P}_{0.1}$ ($\mathcal{P}_r$ with $r = 0.1$) applied in dynamic mode. Unless stated otherwise, we assume a market with $T_{charge} = 1s$ (fractional hours) and zero market signal delay ($\tilde{t_1} = \tilde{t_2} = 0$). Because of the small T_{charge} in the default setting, we have $k = \lceil T_{charge}/T_{opt} \rceil = 1$, $n_t = m_t$, and the MDP state becomes $\{t, w, u, \lceil n_{t-1} \rceil\}$. For optimization $T_{opt} = 1h$ is used by default. When the market has zero signal delay, Cümülön v2 always bids at infinity, otherwise skyline bidding with $c_t = 4$ is used by default.

[4] We collect these information from the simulated training price traces rather than the real historical price data, simply because the latter is too sparse to arrive at full-fledged distributions directly at all price-time combinations, while the former is an enhanced version whose data size is under our control.

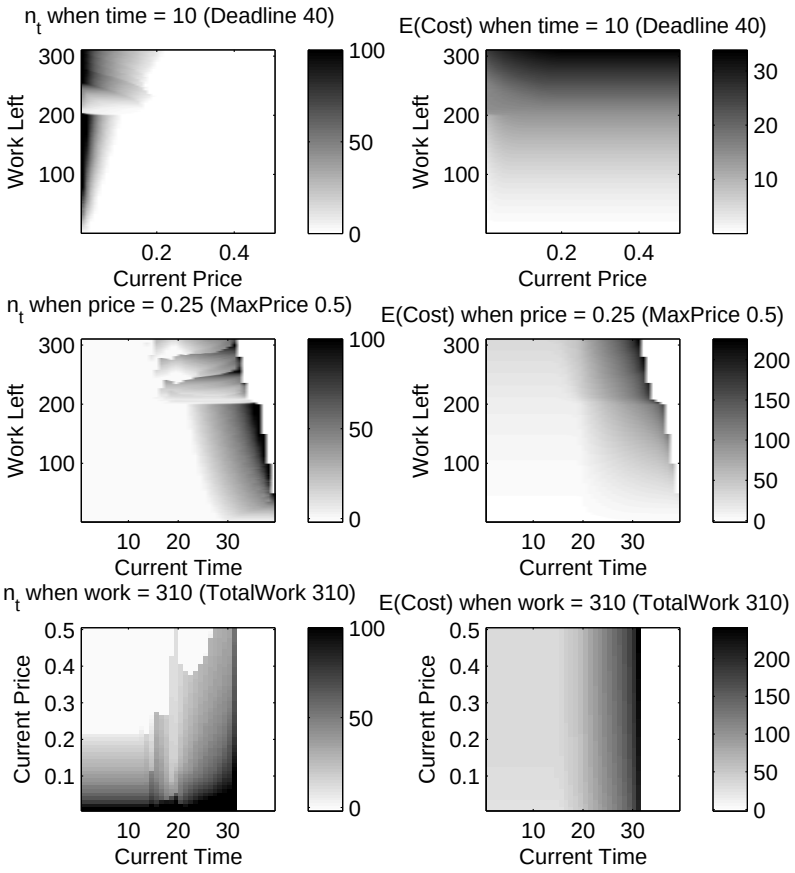

FIGURE 4.1: The proposed optimal choice of $n_t(= m_t)$ and $E(cost)$ at three slices of states, in an artificial three-job workflow of size 310 machine hour. The deadline is set at $40h$.

4.7.1 Visualization of policy $\mathcal{P}_r$

Recall that a policy maps every state $\{t, w, u, [m_{t-1} \cdots m_{t-k}]\}$ to a value m_t. We visualize Cümülön v2's policy $\mathcal{P}_{0.1}$ in Figure 4.1, for illustration purposes, using an artificial workload consisting of three matrix multiplication jobs, with $100, 10, 200$ machine hours worth of work respectively ($w_1 = 100, w_2 = 10, w_3 = 200$), under a deadline of 40 hours. Since the state space is four dimensional ($\{t, w, u, [n_{t-1}]\}$), we picked three slices of states: $t = 10h$, $u = \$0.25/h$ and $w = 310$ respectively with $n_{t-1} = 0$. At each state, the suggested action $n_t = m_t$ (on the left) and the expected cost of the state following the policy $\mathcal{C}(\mathcal{P}, \{t, w, u, [n_{t-1}]\})$ (on the right) are plotted.

As we can see, in the first case with $t = 10h$, since it is still far from the deadline, Cümülön v2 only bid for nodes ($n_t > 0$ on the left) when the market price is low, and that the total expected cost (on the right) depends mostly on w the work left, and less on u the current market price. In the left figure, a trend is that when w increases, Cümülön v2 tends to bid for more nodes and start bidding at higher prices. However, an exception to the trend happens around $w \in (200, 250)$. The reason is that the second job with $w_2 = 10$ is much less scalable compared to the other two jobs, thus the point where getting more nodes no longer benefits is reached at smaller n_t.

In the second case with $u = \$0.25/h$, since the price is relatively high, we should not go for a positive n_t unless either we are too close to the deadline or too much work is left. The upper right zero region corresponds to the states where using a maximum cap of 100 on n_t, we cannot finish the workflow by the deadline. Note that such a cap is introduced only for limiting the search range for $[m_t \cdots m_{t-k}]$ when solving the MDP. The model itself does not require such a cap because the parallelism overhead (recall the speedup function $g(n)$) will naturally limit the optimal choice of n_t. For the same reason as the first case, we see a lower n_t when w is around 200 to 210. Furthermore, since the model only adjust n_t every $T_{opt} = 1h$, this introduces execution waves of length one hour and that's why the trend propagates above to higher w regions. Note that this wave effect also exists in the previous case, but appears less noticeble.

In the third case with $w = 310$ (all work in the workflow), we tend to start bidding for nodes either when the price is lower or when it gets closer to the deadline. When $t > 32h$, we cannot finish the workflow in time given the cap of 100 on n_t. Besides, the left figure reveals impact on Cümülön v2's decision on n_t from another aspect, which is the price transition matrix. The "skyline" (the maximum market price with a positive n_t decision) of the policy becomes higher than usual around $t = 20h$. This

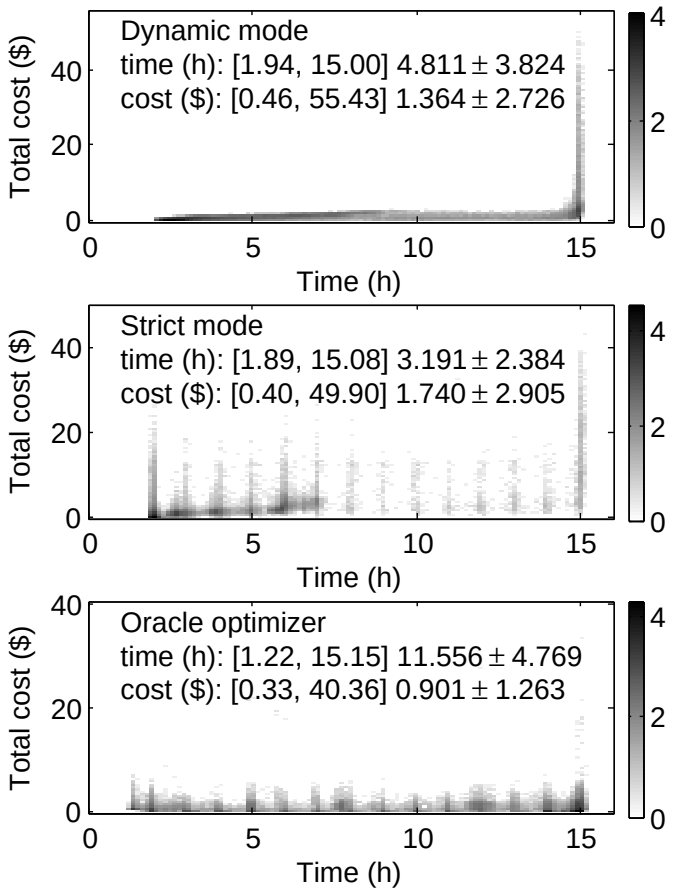

FIGURE 4.2: Simulated total execution time and cost of two iterations of GNMF against 100,000 test price traces, using policy $\mathcal{P}_0$ by Cümülön v2 under a) dynamic mode, b) strict mode, and c) the oracle's plan with strict mode. The deadline is set at 15 hours. The densities are the test trace count in $\log_{10}$ scale. The numbers in the figures are in format: [min, max] mean ± std.

is because our trained price model thinks that there is a larger possibility that the market price will become lower at around $t = 20h$. Since the price transition matrix with this information is incorporated into MDP model, the impact is reflected in the resulting decisions.

4.7.2 Evaluating the plan

Now we evaluate Cümülön v2's policy through simulated execution, using two iterations of GNMF and testing price trace set $S_{0.02}$. The default setting of $T_{charge} = 1s$

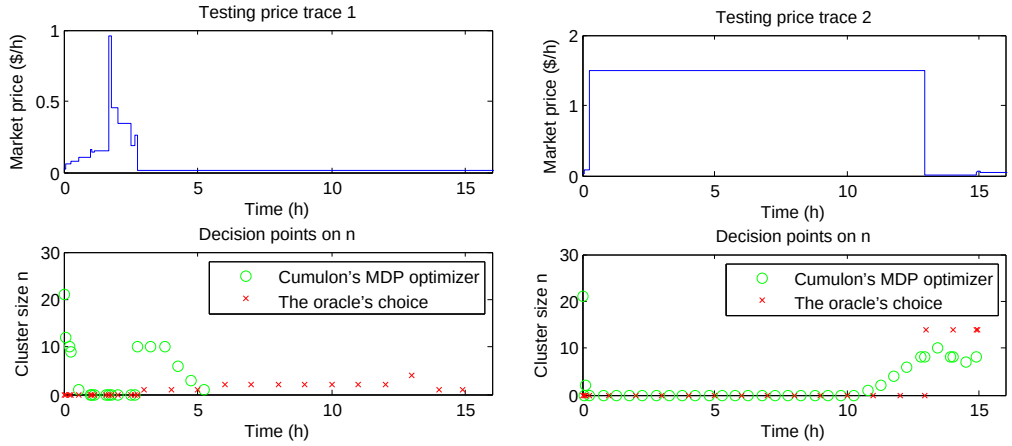

FIGURE 4.3: Two specific price traces in the test set and the corresponding actions took by Cümülön v2 and the oracle in Figure 4.2.

and $T_{opt} = 1h$ is used. Under a deadline of 15 hours, we compare Cümülön v2 proposed policy $\mathcal{P}_0$ that minimizes execution cost, with the plan proposed by the oracle optimizer (described in Section 4.4, with $T_{opt} = 1h$ as well) who foresees the testing price traces and makes the optimal decision minimizing cost for each specific trace. We run policy $\mathcal{P}_0$ under both dymanic mode and strict mode (discussed in Section 4.6), as well as the plan by the oracle optimizer using the execution simulator. The scatter plot of the total execution time and cost against the $100,000$ test traces is shown in $\log_{10}$ scale in Figure 4.2.

As we can see, in dynamic mode (Figure 4.2a), a majority of the densities are in the low cost area with finish time ranging from $2h$ to $15h$. These correspond to the lucky cases where there are enough low market price durations before the deadline for Cümülön v2 to finish the workload with cheap machines. However, there is a distribution of high cost toward the $15h$ deadline. These are the unlucky cases where the market price stayed high and Cümülön v2 chooses to wait for later potential price drop, until the deadline approaches, and Cümülön v2 have to go for the expensive machines to finish in time, incurring a high cost.

Figure 4.2b shows the result of the same policy $\mathcal{P}_0$, but applied in strict mode,

where cluster size n_t is only changed at hour boundaries, same as the $T_{opt} = 1h$ assumption made in the MDP model. As we can see, less densities are concentrated close to the deadline. This is because whenever we get some nodes, they are kept for at least an hour. It is less likely for Cümülön v2 to wait until the deadline to enter the "panic" mode. It also creates the artifact that the finish time tends to cluster close to each hour boundaries. However, since many fine-grained adjustment opportunites are missed, the overall mean of cost increases from \$1.364 to \$1.74.

Finally, the performance of the oracle optimizer is shown in Figure 4.2c. It achieves an overall mean execution cost of \$0.901. Note that the performance of the oracle optimizer is a lower bound, and more importantly, unachievable by any non-oracle model simply because it knows the exact future, while others do not.

We illustrate why this is the case in Figure 4.3, by zooming into the decisions made by Cümülön v2 in dynamic mode and the oracle optimizer in Figure 4.2, during the simulated execution on two selected test traces. Note that both traces start from the initial price $p_0 = \$0.02/h$.

The first test trace (on the left) is one of the lucky cases where prices stay relatively low, although there is a spike at around $2h$. The green circles in the lower figure shows the decisions by Cümülön v2 in dynamic mode. Since Cümülön v2 is unaware of the future, it makes decisions based on its modeled future price distribution. Given the low price ($\$0.02/h$) at time zero, it decides to start with 21 nodes, following the policy from the MDP model. When the price keep increasing within the first hour, it decides to decrease the cluster size n gradually to zero by terminating the nodes (no need to delay the termination since $T_{charge} = 1s$ here). When the price drops to $\$0.016/h$ at around $2.8h$, it resumes the execution with 10 new nodes, and with several more adjustments, eventually finishes execution at around $5.3h$. The incurred execution cost is \$0.72.

The oracle optimizer, however, has chosen a different plan. Knowing that the

price will stay low afer $2.8h$, the best way to minimize cost is to hold execution until then, and more importantly, keep cluster size low and spread the execution to the entire low price region before deadline, so that minimal parallelism overhead (recall the speedup function $g(n)$ in Section 4.2) is incurred. Following the oracle optmizer's plan, the execution finishes right before the $15h$ deadline, and the total cost reduces to \$0.35.

On the other hand, the second test trace on the right of Figure 4.3, represents an unlucky case where the price surges and stay high most of the time. With the same initial price $\$0.02/h$ at time zero, Cümülön v2 starts out with the same $n = 21$ decision. However it have to drop all nodes when price surges soon after to avoid high charge, and start waiting for potential price drop. Until at around $11h$, considering that the deadline is approaching and that there is still a considerable amount of work left, it decides to resume execution with $n = 1$ even when the price stays very high. As the time goes by, the possibility for price to drop in time becomes smaller, and the pressure from a larger n to finish the work in time and thus more parallelism overhead keep increasing. Cümülön v2 decides to keep inceasing n to achieve the best trade-off between the two aspect to minimize the expectation of total cost. Eventually when price does drop in this case at around $13h$, there is only a portion of the work left and only a few nodes are needed to wrap up. Overall, since many nodes are used under expensive price, Cümülön v2's plan incurred a high cost of \$11.99 in this case.

The oracle optimizer, however, is well aware of the exact high price region. It chooses to simply wait, and after the price drop at $13h$, go for a cluster of 14 nodes and finish before deadline. In all, a total cost of \$0.44 is incurred.

Earlier we claim that any non-oracle optimizer cannot do as good as oracle optimizer. The reason becomes quite clear when we compare the decisions across the two price traces in Figure 4.3. For instance, at the inital price of $\$0.02/h$, Cümülön v2 makes the same decision (21 nodes) in both cases, simply because the information

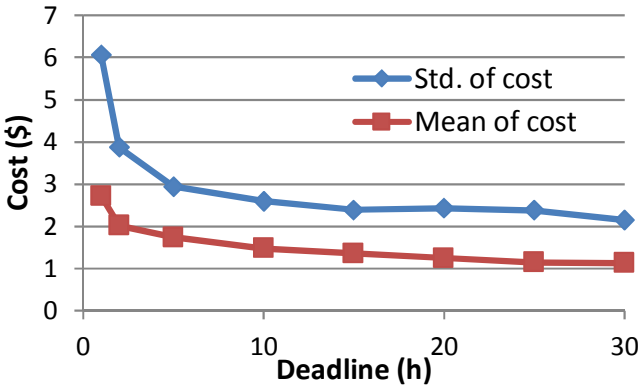

FIGURE 4.4: Mean and standard deviation of simulated cost for two iterations of GNMF achieved by policy $\mathcal{P}_{0.1}$ when user-specified deadline varies.

available by then, including the state it is in and the belief about future price evolvement, are exactly identical. However for oracle optimizer, knowing the full future, we should clearly go for less nodes in the first trace, but many more in the second one.

4.7.3 Impact of deadline

In this experiment we study the impact of the user-specified deadline, using two iterations of GNMF and test set $S_{0.02}$. The result is shown in Figure 4.4. As we can see, both mean and standard deviation of cost decreases when deadline increases. This is simply because as the time we have increases, we can afford to wait for more low price opportunites and finish the work with cheaper nodes. On the other hand, if the deadline is really close, we have to start execution no matter how expensive the nodes are.

4.7.4 Optimization time and quality

Now we show how the quality of the plan for GNMF changes when we vary T_{opt}, and how much time it takes to generate the plan. All other settings are same as above. All optimizations are run single threaded using Java, on a desktop machine with an

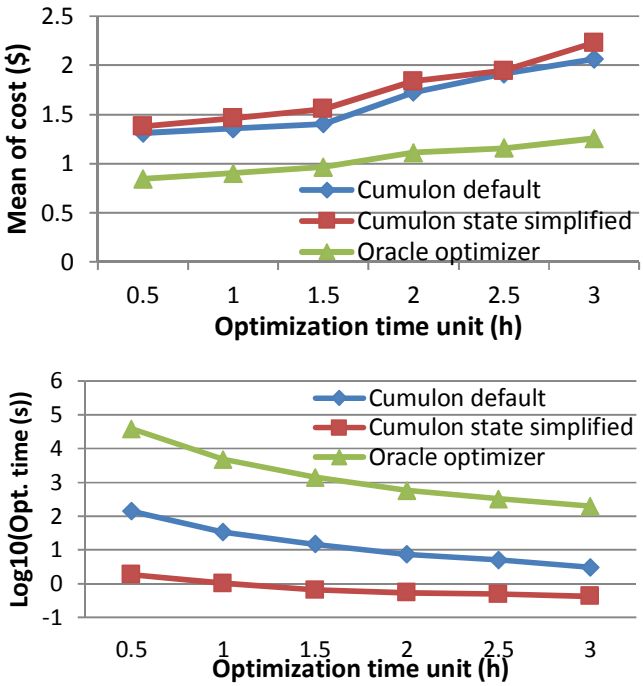

FIGURE 4.5: Average cost achieved by the plan (upper figure) and the corresponding optimization time (lower figure in $\log_{10}$ scale) for GNMF for different T_{opt}. "Cümülön v2 state simplified" is the special case assuming no cluster change overhead, as discussed in Section 4.5.

Intel i7-2600 3.40GHz CPU and 8GB of memory. Besides the default Cümülön v2 applied in dynamic mode as well as the oracle optimizer, we also run the alternative model "Cümülön v2 state simplified" (discussed in Section 4.5), which assumes no cluster change overhead and does not have m_t in its MDP state. The results are shown in Figure 4.5.

In general, as T_{opt} decreases, the optimization becomes more fine grained, leading to a better plan with lower cost. As expected, the oracle optimizer is consistently better than the Cümülön v2 default, which is better than Cümülön v2 state simplified.

On the other hand, the increasing optimization time is the price we pay for the better plans. Note that for both Cümülön v2 optimizers, the time only involves solving an MDP model once. The time for simulation across the testing traces is

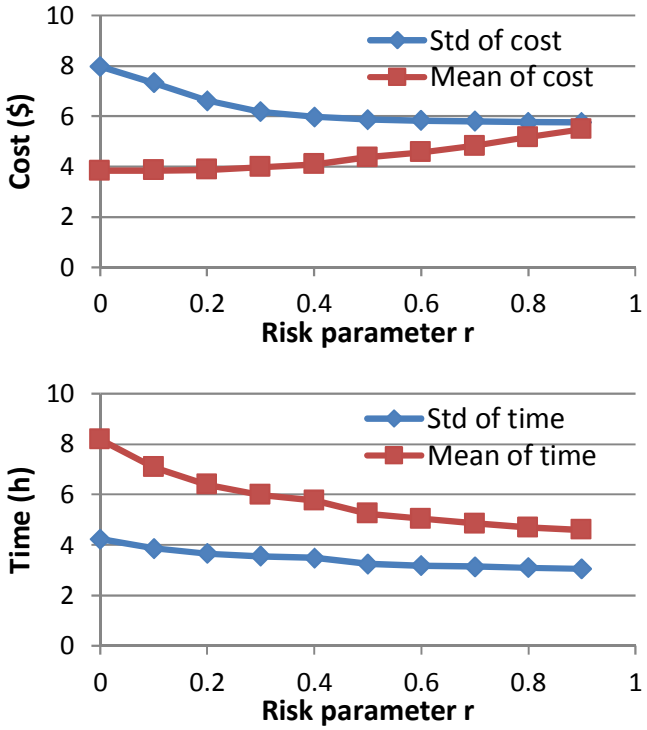

FIGURE 4.6: Mean and standard deviation of simulated cost (upper) and time (lower) for two iterations of GNMF achieved by the policy $\mathcal{P}_r$ when r varies. Test set $S_{0.2}$ used.

not included here. While for the oracle optimizer, it needs to run on each and every testing traces to arrive at a plan, so that its optimization time is proportional to the size of the test set ($100,000$ in this case). Specifically for $T_{opt} = 0.5h$, it took the oracle optimizer around $10h$ to go over all the traces.

4.7.5 Tradeoff between cost mean and variance

In this section we investigate how the performance of policy $\mathcal{P}_r$ changes when we change the risk tradeoff parameter r. Under the default setting of $T_{charge} = 1s$ and $T_{opt} = 1h$, we simulate the policies for two iterations of GNMF under a $15h$ deadline using test set $S_{0.2}$ and plot the results in Figure 4.6.

As we can see, r presents a tradeoff between mean and variance of the cost of the plan. When r increases, more weights are put on minimizing the variance, leading

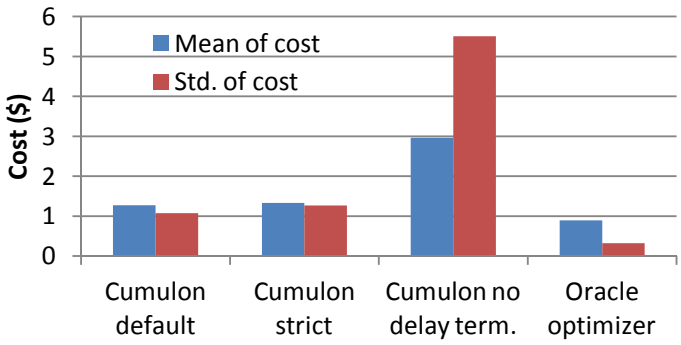

FIGURE 4.7: Mean and standard deviation of simulated cost of RSVD-1 achieved by the different optimizers. A market with $T_{charge} = 1h$ and zero market signal delay assumed.

to a less aggresive plan with lower cost variance but potentially higher cost mean. In general, when in state $\{t, w, u, [m_{t-1} \cdots m_{t-k}]\}$, by choosing a larger m_t, more work is done in the current step and less is left for the underministic future, leading to a plan with less uncertainty and thus lower cost variance. This is why in the lower figure with timing, the more conservative (higher r) the policy is, the earlier execution tends to finish.

Overall, under the test set $S_{0.2}$ where market price starts higher, the cost mean to finish the same workload becomes higher compared with that under test set $S_{0.02}$ as in earlier experiments.

4.7.6 A market that charge by hour

Now we run RSVD-1 in a market with $T_{charge} = 1h$ instead, still asuming a zero market signal delay ($\tilde{t_1} = \tilde{t_2} = 0$), so that Cümülön v2 still bids at infinity. Figure 4.7 shows the mean and standard deviation of the plans respectively by Cümülön v2 in default (dynamic mode) and strict mode, Cümülön v2 without delay termination, and the oracle optimizer.

As we can see, Cümülön v2 in strict mode is still slightly worse than the default dynamic mode, and the oracle optimizer is still the best. Furthermore, to illustrate

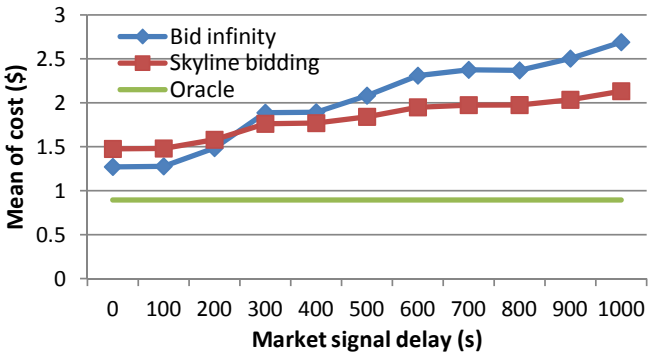

FIGURE 4.8: Impact of market signal delay and the performance of infinite bidding and skyline bidding on RSVD-1.

the importance of delayed termination (discussed in Section 4.6) during execution, we showed the result of disabling it in the third setting. Since $T_{charge} = 1h$ here, whenever a node gets charged, it is charged for the entire next hour. If Cümülön v2 constantly terminates nodes without utilizing its machine time we already paid for, we end up with a very high cost, even with the same optimal policy.

4.7.7 Bidding strategies under positive market signal delay

Now we discuss the cases where the market has a positive signal delay. As discussed in Section 4.5, setting a bid price is a general way to guard against potential unfortunate high charges. Whenever a node is reclaimed because the market price exceeds the bid price, we assume users are not charged for the last partial T_{charge} (hour in this case), just like in Amazon's scheme. (Otherwise, setting a bid price does not make much sense and it would not be benefitial at all to set it.)

In Figure 4.8, we compare Cümülön v2's performance under the infinity bidding and the skyline bidding strategy, with the oracle optimizer, when market signal delay varies. Here we assume $\tilde{t_1} = \tilde{t_2}$. Note the market signal delay will not affect the oracle optimizer at all since it knows the future prices. In fact, the performance of the oracle optimizer here still serves as a theoretic unreachable lower bound for Cümülön v2.

As we can see, the performance of both bidding schemes gets worse as market

delay $\tilde{t_1}$ and $\tilde{t_2}$ increase. This is because Cümülön v2 tends to get nodes when price (p_{t_1}) is low and terminates them when price (p_{t_2}) is high. Since p_{t_1} is usually low, $p_{t_1+\tilde{t_1}}$ is more likely to be higher than p_{t_1} if different. Furthermore, since p_{t_2} is likely high, having to pay for the duration t_2 to $t_2 + \tilde{t_2}$ could be bad. Overall, the actual cost for duration $t_1+\tilde{t_1}$ to $t_2+\tilde{t_2}$ tends to be higher than the cost for duration t_1 to t_2. The larger $\tilde{t}$, the bigger this difference tends to be, and thus worse the performance in terms of cost.

When comparing setting a finite (skyline in this case) bid price as compared with infinite bidding under positive market signal delay, there are two aspects to consider in general. First, it has a positive impact on cost because it guards against the unfortunate case of price surge within the signal delay window. However secondly, setting a non-infinite bid price also has a negative impact on cost. Since we cannot change the bid price on nodes we own, they might be reclaimed later even if we become willing to pay a higher price to keep them.

As we can see in Figure 4.8, when the signal delay is smaller than four minutes, the negative impact dominates the positive one, and thus skyline bidding is actually worse than infinite bidding. As the signal delay gets larger, the positive impact of setting a finite bid price increases, and eventually dominates the negative one. The skyline bidding becomes better than infinite bidding scheme.

4.7.8 Evaluating the skyline bidding

In this experiment, we fix the market signal delay at ten minutes, and compare Cümülön v2's skyline bidding with simple fixed-price bidding schemes. Specifically, we run the same policy suggested by the MDP model, except that different strategies (skyline vs. fixed price) are used to set bid prices. Note that during the last T_{opt} before the deadline, a bid price of infinity is always applied regardless of the strategy to ensure meeting the deadline. The results are shown in Figure 4.9.

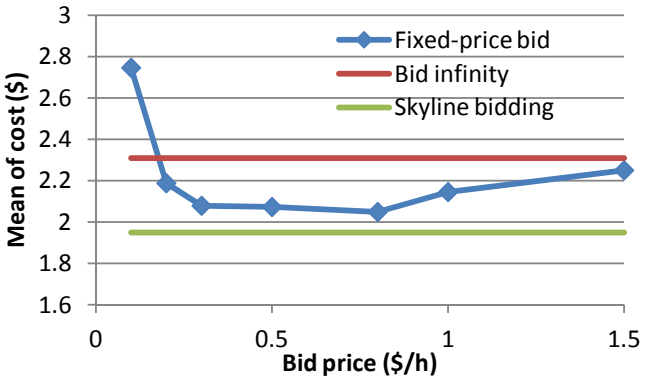

FIGURE 4.9: Comparing skyline bidding to fixed-price bidding under market signal delay of ten minutes for RSVD-1.

As we can see, when the bid price is fixed at low values, the performance is bad. This is simply because the bid price is too low and we cannot get any nodes even if the policy think we should. When the bid price applied increases, this conflict gets alleviated and the average cost decreases. However when the bid prices gets further higher, the performance gets worse again, and eventually converges to the bid infinity case. This is because the bid price becomes too high compared to the desired threshold implied in the policy. The safety bar against price surge is too high.

More importantly, the performance of the skyline bidding outperforms the all fixed-price bidding schemes. This is because the "skyline" (maximum market price with non-negative action m_t) of the policy, as illustrated earlier in Figure 4.1, is a moving target that changes through time. Consequently, a fixed bid price cannot accommodate the need while Cümülön v2's skyline bidding strategy can.

4.7.9 Comparing with existing approaches

Many existing works tried to analytically derive the optimal bidding price by modeling the market price and the workload. We refer to them as fixed bidding schemes, where they keep bidding at fixed bid price and bid size (cluster size n) throughout

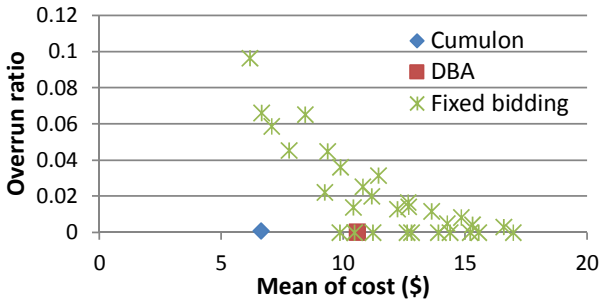

FIGURE 4.10: Performance of Cümülön v2 in comparison with existing approaches: fixed bidding and DBA.

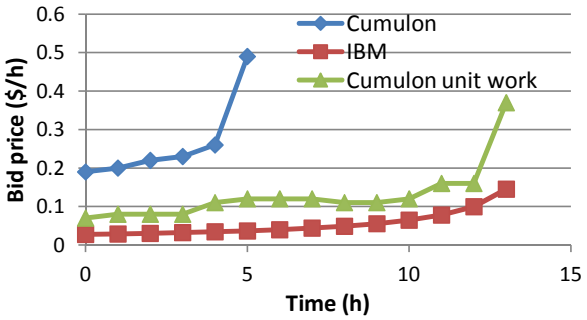

FIGURE 4.11: The dynamic bid prices used by Cümülön v2 and DBA for the RSVD-1 workload in Figure 4.10. Additionally, Cümülön v2's bid prices for an imaginary unit-size workload is also shown for comparison with DBA. For each line, the unplotted points before the $15h$ deadline means a bid price of infinity.

the execution until workload is finished. Besides fixed bid price approaches, [47] proposed a dynamic bidding algorithm (called DBA) that adjusts the bidding price on the run. For bid size, they assume perfect speedup for the workload, i.e. $g(n) = n$, so that the optimal plan for a workload with w amount of work is equivalent to that of an unit-work workflow, except that the bid size is w times larger. Consequently, they always use a bid size n large enough that the workload can be finished within one time step (hour).

In this experiment we compare Cümülön v2 with these existing approaches using RSVD-1. We use $T_{charge} = 1h$ (as assumed by DBA) with a $15h$ deadline. To cover all the potential optimal plans by various fixed bidding results, we tried various

combinations of reasonable bid prices (from \$0.05/h to \$1/h) and bid sizes (from 5 to 80). For DBA, we used a fixed bid size of $n = 60$ nodes because using 60 nodes, RSVD-1 finishes in less than an hour with high probability.

Similar to Cümülön v2, we enforced a bid price of infinity in the last time step before the deadline in both approaches, in order to meet the deadline with higher chance. Figure 4.10 shows the result in terms of cost mean achieved and the overrun ratio, i.e. the ratio of testing traces in which execution does not finish by the deadline. Test set $S_{0.2}$ is used here.

As we can see, Cümülön v2 clearly out performs the other two approaches. If we look at the fixed bidding plans, they presents a tradeoff between meeting the deadline and cost mean. Plans with lower bid prices bid sizes tend to have a cheaper cost (because of higher average node price and higher parallelism overhead respectively), but at the same time are less likely to meet the deadline. A relatively good plan achieves a mean cost of \$9.87 and a near-zero overrun ratio, by keep bidding at \$0.5/h with $n = 20$. In general, fixing the bid price and bid size without considering the state, including market price then, amount of time and work left, is quite limited compared to Cümülön v2's dynamic bidding. That's why Cümülön v2 does a much better job and achieved a mean cost of \$6.62. Furthermore with Cümülön v2, it is very unlikely to miss the deadline simply because its ablility to adjust its decisions dynamically according to the progress.

The DBA algorithm followed a very similar approach to our MDP model. However, because of the perfect speedup assumption, they fixed the bid size at 60 nodes, incuring lots of parallelism overhead in reality. That's why even with dynamic bidding, it only achieves a mean cost of \$10.59, even worse than some fixed bidding plans. Furthermore, as the size of the workload further increases, or if T_{charge} gets smaller, DBA will have to go for even larger cluster to push everything into one time step. Its performance will be even worse because of the parallelism overhead, and

eventually becomes invalid because the workload is simply not scalable enough to be squeezed into a single time step.

Now back to this example, Figure 4.11 visualized the specific bid prices used by Cümülön v2 and DBA respectively. Since DBA always finishes the workflow within one time step, they only need to succeed in getting the nodes once. As a result, their bid prices are lower, and depend only on market price model but not the workload at all. On the contrary, Cümülön v2's skyline bid prices considers both market price evolvement as well as the workload information, since it is derived from the MDP model. In the case of RSVD-1 with $15h$ deadline, it starts bidding higher in very early stages, so that execution can be spread out into the $15h$ time range with small cluster sizes.

Besides, we also run Cümülön v2 against an imaginary unit-size workflow (a workflow that takes one machine-hour to finish), and plot the suggested bid prices as well. As expected, it becomes very close to the DBA's decision. But it is still slightly higher than DBA. This is because to analytically derive the DBA algorithm, [47] made another unrealistic assumption that the market price of the spot instances would never be higher the on-demand price. This is far from true, and Cümülön v2's market price model is trained with real spot price data. This is where the gap comes from.

4.7.10 Comparing Cümülön v2 with Cümülön v1

Finally we compare Cümülön v2 with Cümülön v1. Briefly, Cümülön v1 tackles the problem by using a combination of on-demand and spot instances. First respecting the deadline, a baseline plan using only on-demand nodes is generated. Next, a single batch of spot instances are acquired with identical bid prices, trying to speedup the execution and reduce cost. It is a independent solution without any external assumptions. While in Cümülön v2, we assume there is an external reliable storage

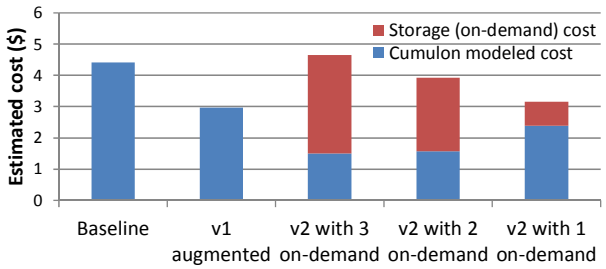

FIGURE 4.12: Comparing Cümülön v2 using on-demand instances to run the external storage with Cümülön v1. Another RSVD-1 workload (with $l = 2048$, $m = n = 51,200$, $k = 15$) is used.

service available. The fundamental difference in the system setup makes it hard to do a fair comparison.

Now we ask: what if we run several on-demand instances and let them be the external storage service required by Cümülön v2, what would the comparison look like?

Here we used a longer RSVD-1 workload with reduced matrix size, and run both system using a $10h$ deadline, using test set $S_{0.02}$. The baseline plan of using three on-demand nodes has a cost of \$4.42. Cümülön v1 enhanced the baseline by bidding for 64 spot nodes at bid price \$0.08/h. The overall expected cost reduces to \$2.96.

While for Cümülön v2, we tried using one, two and three on-demand instances for storage, and plot the costs for compute (spot nodes) and storage (on-demand nodes) separately. As the number of on-demand storage nodes decreases, on one hand the stoarge cost (red part) decreases, but on the other hand writing data to it becomes slower, leading to a longer execution time and larger cost for spot nodes.

If we only look at the modeled compute cost of Cümülön v2, especially with a scalable storage service with minimal writing overhead, it out performs Cümülön v1 significantly. So when we happens to have a "free" HDFS on the side, or there is a cloud provided storage service with neglegible cost (\$0.04/GB/month for Amazon S3), Cümülön v2 is a better choice. However, if there is no better choice and we

need to use spot instance for storage, then Cümülön v1 could be better.

4.8 Conclusion

In this chapter, we presented Cümülön v2, our second approach for supporting spot instances in the context of Cumulon. Under the assumption of external reliable storage, Cümülön v2 maintains a single pool of spot instances for computation, and dynamically adjusts the pool size during execution considering the market price, time to deadline and the amount of work left. We model the problem of making decisions on discretized future states as a Markov Decision Process (MDP) and solve it offline, and then apply the plan dynaminically during execution. Under a positive market signal delay, a proper bid price is needed to guard against potential high charge brought by sudden price surge. We proposed a skyline bidding scheme based on the MDP model that dynamically suggests the bid price to set for new spot instances according to the MDP state during execution.

Although developed in the context of Cumulon, the models and solutions developed for spot instances are generally applicable to other scalable parallel workloads and systems, as long as the speedup function and the cluster changing overhead term associated with the workload are provided. We leave these generalizations to future work. Currently Cümülön v2 only considers spot instances for computation, even when the market price remains very high. It might be better to fall back to on-demand instances under these cases. As future work, it would be interesting to investigate how to extend the MDP model to use a mixture of on-demand and spot instances.

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 these, we have presented Cumulon, an end-to-end system 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 thesis, 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.

5.1 Future Work

As discussed in Section 3.3, 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 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 [31]. It would be nice to support both forms of parallelism. *SystemML* has made promising progress in this direction [4], 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.

Appendix A

Proof of the greedy algorithm in Chapter 4.4

Under the assumption of deterministic price, $T_{charge}|T_{opt}$ case with $o(n_1, n_2, w) = 0$, the optimization problem becomes: $\min \sum_{t=1}^{T} n_t c_t$, s.t. $\sum_{t=1}^{T} g(n_t) \geqslant W$. Here $T = D/T_{opt}$, $c_t = Charge(t)$, both are given constant. Now using the greedy algorithm of keeping incrementing n_t with the t that minimizes $(g(n_t + 1) - g(n))/c_t$, we prove that during every step of the algorithm, the set $\{n_t\}$ is the optimal solution for the subproblem with $W = \sum_{t=1}^{T} g(n_t)$.

For the base case of $\{n_t = 0\}$ for all t, the claim is clearly true. Next assume the algorithm has arrived at set $\{n_t \geqslant 0\}, t = \{1, \cdots, T\}$, and that $V = \sum_{t=1}^{T} n_t c_t$ is the minimal cost achievable under constraint $W = \sum_{t=1}^{T} g(n_t)$. Without the loss of generality, assume in the next step the algorithm picked n_1 and incremented n_1 to $n_1 + 1$. Now with

$$W' = g(n_1 + 1) + g(n_2) + \cdots + g(n_T)$$

we have new cost

$$V' = (n_1 + 1)c_1 + n_2 c_2 + \cdots + n_T c_T = V + c1$$

Since n_1 is picked in this step, we have for all t:

$$g(n_t + 1) - g(n_t) \leqslant \frac{c_i}{c_1}(g(n_1 + 1) - g(n_1))$$

Assume there is another choice set $\{n'_t \geqslant 0\}$, such that $\sum_{t=1}^{T} g(n'_t) \geqslant W'$. Define $\Delta n_t = n'_t - n_t$. We simply need to prove that $\sum_{t=1}^{T} n'_t c_t \geqslant V'$, or equivalently $\sum_{t=1}^{T} \Delta n_t c_t \geqslant c1$.

Since the speedup function $g(n_t)$ is concave, we have

$$\begin{aligned} g(n_t - 1) - g(n_t) &\leqslant g(n_t) - g(n_t + 1) \\ g(n_t - 2) - g(n_t - 1) &\leqslant g(n_t - 1) - g(n_t) \leqslant g(n_t) - g(n_t + 1) \\ &\vdots \end{aligned}$$

If $\Delta n_t < 0$, by adding above inequalities together, we have

$$g(n_t + \Delta n_t) - g(n_t) \leqslant \Delta n_t \left[g(n_t + 1) - g(n_t)\right]$$

Similarly, the same inequality are be derived when $\Delta n_t \geqslant 0$. Now:

$$\begin{aligned} g(n_1 + 1) - g(n_1) &\leqslant \sum_{t=1}^{T} g(n_t + \Delta n_t) - g(n_t) \\ &\leqslant \sum_{t=1}^{T} \Delta n_t \left[g(n_t + 1) - g(n_t)\right] \\ &\leqslant \sum_{t=1}^{T} \Delta n_t \frac{c_i}{c_1} \left[g(n_1 + 1) - g(n_1)\right] \end{aligned}$$

Since $g(n_1 + 1) - g(n_1) > 0$, we have $\sum_{t=1}^{T} \Delta n_t c_t \geqslant c1$. ▫

Appendix B

Spot Price Simulation Model and Validation

Disclaimer: The work in Appendix B is done primarily by our collaborator Nicholas W.D. Jarrett*, who was then a PhD student in the Department of Statistical Science at Duke University. We include this material here for the sake of completeness.*

In this section we explain in detail the spot price simulation model developed for Cümülön v1, and then provide discussion and results on the model validation.

B.1 The Price Model

We specify the fundamental properties of the model we used to forecast spot-prices. We provide commentary on how this model is built upon and expands existing methodology, as well as how this impacts validation. We start with a brief literature review.

B.1.1 'Dynamic Resource Allocation for Spot Markets in Clouds,' Qi Zhang et. al. USENIX

Spot-price forecasts are obtained through modeling supply and demand. Amazon's auction mechanism is viewed as a (single round) continuous seal-bid uniform price

auction with adjustable supply, implying that end-users' optimal strategy is to bid their honest valuation of the resource(s) under consideration. However, after observing multiple rounds of Amazon re-selling their excess capacity as spot-instances an astute end-user could end up bidding a value other than their truthful valuation of their requested resources. This is one of several self-admitted limitations of this research, and is in fact suggested as a direction of future research.

Demand is modeled with a simple auto-regressive (AR) model of a fixed order with zero mean, white-noise distributed errors. While demand is theoretically continuous, the AR model is discrete based on a regular partitioning of the requested range $[t_0, t_0 + T]$. Utilizing the single-round view of the auctioning system as explained above, demand can be modeled independently at each possible bid price.

Based on a demand curve, supply can be determined through revenue optimization from the perspective of Amazon. The revenue optimization is taken from the perspective of a monopoly market, however this assumption has been targeted for relaxation in future extensions of this paper. Unfortunately, the exact objective function is quite difficult to solve, being at least as difficult as NP-hard, as a result this problem is solved approximately.

B.1.2 'Achieving Performance and Availability with Spot Instances,' Michelle Mazzucco & Marlon Dumas IEEE

It is stated that spot prices do not appear to follow any particular law, and that the series may have nearly zero correlation. The investigation of auto-correlation of the price series over time is taken discretely over time, similar to the discrete AR model for the continuous underlying demand process in the previous paper.

The proposed algorithm depends on whether or not the recent price series exhibited lag l auto-correlation of sufficient magnitude so that extending the recent trend into the future will reliably be on-target.

- If the correlation is at least 0.4 in absolute value, then forecast prices using regression

- Otherwise, approximate prices with a Normal distribution, and use the Normal quantiles to inform your bid price. (Extreme value distribution)

B.1.3 'Statistical Modeling of Spot Instance Prices in Public Cloud Environments,' Behman Javadi et. al. IEEE

This paper forecasts spot-prices by forecasting two time series, one for the value of the price at a given time, and another for the length of time that each price persists for.

Statistics on the marginal distributions of price and the length of time each price persisted for, inter-price times, are investigated separately. The data indicate that direct approximation with a Normal distribution may be sufficient only as a first-order approximation, with both series showing non-zero skew and excess kurtosis (which is inconsistent with the Normal distribution). Relatively speaking, the price distribution is closer to Normality than the inter-price time distribution, exhibiting less skew and excess kurtosis.

They also wrap time around a one week cycle, and demonstrate interesting day-of-week and time-of-day trends apparent in spot-price markets. Prices exhibit time-of-day patterns with increasing trends over the first half of each day and decreasing trends over the second half of each day in the EU-West data center. Additionally, day-of-week patterns generally indicate minimal prices on weekends, with prices increasing as the next set of weekdays appear. Depending on the data center, the strength of this pattern may differ and there may be a tendency for maximal prices to be achieved on a specific day.

A mixture of of Gaussian distributions with either 2 to 4 components is used to approximate the distribution of prices and inter-price times. The mixture parameters

were determined via the popular Expectation Maximization (EM) algorithm, and Model Based Clustering was used to assess the best model in terms of the number of components as well as to aid in the parameter fitting procedure. Lastly, several methods are used to assess the quality of the fit of the mixture distributions:

- Visual assessment of probability-probability 'PP' plots

  In a PP plot, two cumulative distribution functions are plotted against one another as their argument is varied from $-\infty$ to $+\infty$. Therefore, the distributions are identical if and only if all points lie on the line which connects the points $(0, 0)$ and $(1, 1)$.

  In other words a PP plot is given by the curve of points $\{(F(x), G(x)) \mid x \in (-\infty, \infty))$ or an approximation there-of if $G(x)$ is taken to be the empirical CDF of a dataset.

  This is not to be confused with the popular QQ plot in which the quantiles of two distributions are plotted against each other as the underlying percentile is varied from 0 to 100. One obvious distinction is that the x and y variables are not constrained to lie in $[0, 1]$ as they would be in a PP plot. Like PP plots, the distributions are identical if all points lie on the line which connects the points $(0, 0)$ and $(1, 1)$; however it is important to consider whether the variables have been transformed. Usually the x-variable of a QQ plot corresponds to a theoretical distribution and the y-variable is the empirical quantiles of a data set.

  In other words a QQ plot is the given by $\{(x_f, y_g) \mid \int_{\infty}^{x_f} f(x)dx = p, \int_{\infty}^{y_g} g(x)dx = p, p \in [0, 1]\}$ or an approximation there-of if y_g is taken to be the empirical quantiles of a dataset.

- Kolmogorov-Smirnov test and the Anderson-Darling test

Nonparametric tests of the equality of two continuous, one-dimensional distributions.

The one-dimensional K-S test statistic is given by the supremum of the difference of the cumulative distribution functions. While this is generalizable to higher dimensions, high-dimensional cumulative distribution functions can depend on the path taken. For a trivial example, consider taking conjunction of the cumulative distribution function of x and the survival function of y.

The Anderson-Darling test also relies on a statistic based on the cumulative distribution function; however it is less sensitive to the issue of path choice described above. This is because it the test statistic is given by integrating the squared difference between the empirical CDF and the null CDF divided by the null CDF times the null survival function. Recall that the survival function is 1 minus the cumulative distribution function.

- Neither test is immediately equipped to validate time-series data however, although the time-marginalized distribution of a univariate time-series could be readily analyzed with either test, and with some care, the bivariate time-marginalized distribution of price and inter-price time could be assessed as well.

B.1.4 The Model

We model forecast spot-prices via forecasting two time-series independently given the recent spot-price history

- A series of the magnitude of the jumps between successive spot-prices
- A series of 'interarrivals' which is given by the duration each price will persisted until updating.

For simplicity refer to the series of magnitudes between prices as *dprices*, and the series of the length of time that each price persists for as *itimes*. The original observed trace of the spot-price over time is given by the bivariate pair of the cumulative sum of each of these series, starting at the current time and price (t_0, p_0).

Exploratory data analysis indicated that itimes and dprices are not strongly dependent on one-another, and thus we forecast them independently given the recent history of market prices. As a result, we can combine the non-parametric density estimates of dprices and itimes with inverse transform sampling to forecast each time-series and then recombine to obtain forecasts on the time domain of the observed spot-price series. If these independence relationships did not hold, inverse transform sampling would be less attractive as it would require higher dimensional integrals which would be significantly more difficult to work out.

Validating evidence is presented for both series independently, as well as for the forecasts of spot-price given by the cumulative sum of the bivariate pair of time series to further lend weight to the assertion that modeling dprices and itimes as conditionally independent is sufficient to produce useful forecasts of spot-price.

Both dprices and itimes are forecast into the future using densities which are approximated non-parametrically with kernel methods, and where the data is given by the recent history of spot-prices wrapped around a 1 week cycle starting on Monday 12:00:00AM and ending on Sunday 11:59:50PM. More specifically, we chose to work with the Epanechnikov kernel for several reasons.

- The Epanechnikov kernel is optimal in a mean squared error sense.
- The Epanechnikov kernel is bounded, so we will have zero probability of sampling radical price changes and time intervals.
- The Epanechnikov can be easier to integrate than many other popular choices which might rely on non-trivial normalizing constants. For example, consider

the standard Epanechnikov kernel (bandwidth=1)

$$k(x) = \frac{3}{4}\mathbb{1}\{|x| < 1\}(1 - x^2) \tag{B.1}$$

$$\begin{aligned} K(x^*) &= \int_{-1}^{x^*} k(x)dx \\ &= \frac{3}{4}x - \frac{3}{4} \cdot \frac{1}{4}x^3 \Big|_{-1}^{x^*} = \frac{3}{4}x^* - \frac{1}{4}(x^*)^3 + \frac{1}{2} \end{aligned} \tag{B.2}$$

Recall that the kernel density estimate of an unknown density f is given by

$$\hat{f}_h(x) = \frac{1}{nh}\sum_{i=1}^{n} k\left(\frac{x - x_i}{h}\right) \tag{B.3}$$

Where h indicates the bandwidth, and generally should be chosen depending on the granularity of the data. Small values of h are theoretically good to obtain high accuracy estimates of the density being studied, however in a finite sample, small values of h can lead to excessive bumpiness of the estimated density. Sampling from the kernel density estimate approaches sampling from the observed data with replacement (i.e. bootstrapping) as $h \to 0^+$.

$$\begin{aligned} \hat{F}(x^*) &= \int_{-\infty}^{x^*} \hat{f}_h(x)dx = \int_{-\infty}^{x^*} \frac{1}{nh}\sum_{i=1}^{n} k\left(\frac{x - x_i}{h}\right) dx \\ &= \frac{1}{nh}\sum_{i=1}^{n} \int_{-\infty}^{x^*} k\left(\frac{x - x_i}{h}\right) dx \\ &= \frac{1}{nh}\sum_{i=1}^{n} K\left(\frac{x^* - x_i}{h}\right) \end{aligned} \tag{B.4}$$

Equation (B.2) implies that $\hat{F}(x^*)$ can be efficiently computed in closed form and without complex normalizing constants. Note for example that whenever $|x^* - x_i| > h$, then $K\left(\frac{x^* - x_i}{h}\right)$ is either 0 or h depending upon whether or not $x^* < x_i$.

For inverse transform sampling, we still need to invert $\hat{F}(x^*)$. This can be easily achieved via numerical approximation, however we can find local inverses exactly by capitalizing on the fact that $\hat{F}(x^*)$ is locally a cubic polynomial in x^* when we are on the observed support modulo the bandwidth. The definition of 'locally' here is the set of x^* for which $\{i|x^* - h \leqslant x_i \leqslant x^* + h\}$ remains unchanged. On this set, the contribution of each x_i is a fixed equation in x^*, and in specific each $x_i > x^* + h$ contributes $1/n$ to $\hat{F}(x^*)$, each $x_i < x^* - h$ contributes 0, and the contribution for all other x_i is a fixed cubic polynomial as discussed above. Since $\hat{F}(x^*)$ is globally monotone increasing, this procedure yields the correct inverse on the interval from $[\hat{F}(x_0^*), \hat{F}(x_1^*)]$ where x_0^*, x_1^* are respectively the minimum and maximum of the local set of x^*.

The convergence of $\hat{f} \to f$, and in general the consistency of kernel density estimators often depends on an infinite sequence of bandwidths which tends towards zero as more data is collected, however the convergence to zero of the bandwidth should generally be slower than the rate at which data is collected in that $nh \to \infty$. There are many papers and guidelines for optimal bandwidth selection, for an introduction see, 'Bandwidth selection for kernel distribution function estimation,' Altman and Leger (*Journal of Statistical Planning and Inference*).

To summarize, our method to draw simulations of forecasts of the spot-price is given by

1. Starting at the initial time and price (t_0, p_0), and then for each subsequent current time and price (t_c, p_c)
2. Draw $u_1, u_2 \sim \mathsf{Uniform}[0, 1]$
3. Draw the next element of dprices and itimes by computing $\hat{F}^{-1}_{\text{dprices}}(u_1)$ and $\hat{F}^{-1}_{\text{itimes}}(u_2)$ via on numerical approximation or exact local inverse based on cubic polynomials on the 'local' set corresponding to u_1 and u_2 respectively.

Table B.1: Summary Statistics of Time-Marginalized Series.

	Min	Max	IQR	Median	Mean	Skew Ex.	Kurtosis
dprices	-4.83	4.96	0.004	≈ 0	0.001	0.01	44.4
itimes	137(s)	24.16(h)	9.78(m)	5.82(m)	58.9(m)	5.33	28.3

Where $\hat{F}_{\text{dprices}}$ and $\hat{F}_{\text{itimes}}$ are the estimated cumulative distribution functions of the magnitude between t_c and the next future price, and the length of time that price will persist for estimated based on the historical data wrapped around a one week cycle.

4. Repeat Steps 2-3 until the sum of itimes is at least T

5. Take the cumulative sum of the time series of ordered pairs (itimes,dprices) starting from the current time and current price (t_0, p_0) to obtain one forecast trace of spot-price.

See Table B.1 for information on basic statistics of the time-marginalized set of observations in dprices and itimes. These are theoretically important for validation since the limiting distribution of the fitted time-series model should roughly match these empirical quantities. These statistics are based on about 9000 sequential price updates, and sample size will impact the distribution of these quantities, with most simply increasing in accuracy as sample sizes increase; however for quantities like the minimum and maximum, their location will become more extreme as sample sizes increase in that their expectations will polarize to the ends of the support of the 'true' distribution of the data. Note that the due to the wide range of itimes, we vary the units for different quantities so they can be more readily interpreted (e.g. exactly how long is 100000 seconds).

B.1.5 Comparisons with Extant Forecasting Methods

Our use of non-parametric density estimation to dynamically change the distribution of the innovations of sequential elements of dprices and itimes gives us some

similarity with the prediction procedure from Section B.1.2 in that our forecasting can automatically transition between forecasting which resembles regression based on time-varying varying parameters, and forecasting based on underlying white noise processes.

Additionally, the paper from Section B.1.2 cites potential issues in their approach with matching the shape and tail of the distribution, although they state that the Normal approximation is generally sufficient (specifically for price prediction). This interesting structure is identifiable by our proposed non-parametric approach, at the cost of needing to search a significantly larger model space to attain a high accuracy density.

There are some important similarities and differences between our approach and those directly based on AR models, such as the model referenced in Section B.1.1.

- The non-parametric model proposed leads to trajectories which never diverge from the range of prices actually observed in the training data. Depending on the values of the auto-regressive parameters, AR series can also have a strong tendency to return to a 'mean' value. Such parameter combinations are ideal for the spot-price series, since it is characterized by rapid return to 'standard' market prices after short bursts of volatility. Graphs supporting this characterization of the spot-price series are given later as we investigate the dprices series.

  - For a trivial example consider the AR(1) series which starts at 0, and is innovated by $N(0, 1)$ at each time-step.

    At time T, the distribution of the value of this series is given by $N(0, (1 - \phi^{2T}\phi^2)/(1 - \phi^2))$ and limits to $N(0, 1/(1 - \phi^2))$ as $T \to \infty$ when the auto-regressive parameter ϕ is less than 1 in absolute value. Note that the variance is finite arbitrarily far into the future.

If the series is known to be at value v at time t, then at time T the distribution of the value of the series is given by $N(v\phi^{T-t}, (1-\phi^{2(T-t)}\phi^2)/(1-\phi^2))$ which again limits to $N(0, 1/(1-\phi^2))$ as $T \to \infty$ providing that the auto-regressive parameter ϕ is less than 1 in absolute value.

- In an AR model, a one-time shock generally always effects the future values of the series infinitely into far the future, whereas this behavior does not necessarily manifest for our non-parametric approach, due to the wrapped distribution over time.

  - This behavior can also be seen in the simple example given above, where knowledge that the value of the series is v at time t implies that all future values will be greater in expectation by $v\phi^{T-t}$ for $T \geqslant t$ compared with the case where the series is known to be 0 at time t.

- Both approaches can make use of the immediate price history

  - However our approach down-weights the previous prices if they occurred far back in time. This is a real concern since the itimes distribution is heavy tailed, with sometimes an entire day going by without the price updating.

    This type of down-weighting similarly happens with large price differences, where for example a large positive price spike is likely to be followed by a large negative price spike, and not the fine-tuning which is typically observed at low and medium price levels. This characterizes exactly how the non-parametric approach is able to capture regression towards the mean, which AR models also capture. The difference is that this is a feature which is built into an AR model providing that it is parameter constrained, whereas the non-parametric model has to learn this feature

of the series.

  - Our approach also combines the immediately recent prices with those observed at the same time the week before.

In regards to the paper summarized in Section B.1.3, there are several similarities originating from their use of mixture modeling, and wrapping data along a one-week cycle.

- The observed prices are wrapped around a one-week cycle, and analyzed for time-of-day and day-of-week trends. Such patterns are discoverable within the non-parametric framework we have proposed, where data points which are nearby the current time (wrapped distribution - modulo 1 week) and price combination will add weight in the kernel density estimates for the next innovation of the spot-price series.

  - In specific, they divide each 24 hour period is split into 8 3-hour periods. This granularity was chosen due to the fact that that the minimum itime in most data centers was about 1 hour - however the minimum itime for our data is around 5 minutes. This is not inconsistent with their findings, since they show that starting in mid-July, activity in the spot-price market dramatically increased, and state that for the US-East data center, this transition occurred in August 2010, which predates our data.

  - For example, in the EU-West data center, prices exhibit time-of-day patterns with increasing trends over the first half of each day and decreasing trends over the second half of each day. Furthermore, it is shown that prices exhibit day-of-week patterns which may vary by data center. For EU-West, prices tend to decrease on weekends; however for other data centers maximal prices have been observed on Tuesdays with minimal

prices on Saturdays and increasing prices on Sundays. Furthermore, the authors state that these patterns are more significant in US-East, which is the data center we are currently working in.

- Spot-price and itimes are modeled using a mixtures of Gaussians whose parameters and number of components are identified using EM and MBC (model based clustering). There is no specific mention of cross-correlation functions,

  Based on this it would seem that the price series and the inter-price times series are forecast independently, much like they are in our proposed approach.

There is a minor amount of ambiguity with respect to the exact nature of model calibration and forecasting. For example, the variable h appears to be used before instantiation in line 9 of Algorithm 1 for model calibration. Despite the day-of-week and time-of-day trends which are observed in the earlier sections of the paper, the mechanism by which the mixture of Gaussian distributions captures these trends is unclear. If these are not explicitly forecast, then this could be a potential strength of our proposed approach.

However, the model calibration section appears to take into account that the weights on the components of the Gaussian mixture model may change across months, which represents a relative strength of their proposed method. Since their analysis is performed on about one year worth of data, parameters within components of the normal distributions within the Gaussian distributions are likely shared across months, with only the weights of these distributions varying dynamically.

This information sharing is central to why we did not extend our non-parametric approach to a month-of-year scale since we would need to go back a full year to gain another single observation of the spot-price behavior in a given one month period; however we could certainly take into account somewhat higher order trends without spreading the data too thin.

B.2 General Comments, A Path to Validation

In order to validate the spot price model, we would like to show that the density we discover for evolving prices forward through time is both a close match to both the training data, and to the observed market states in the immediate future. To evaluate the second point, the training data was split into two sets - one to train the distribution to use in the validation procedure and one test the distribution against. This is to simulate the fact that in a real-life situation the historical price data would be leveraged to generate predictions of the behavior over the next few hours to days - depending on the volume of the workload.

Furthermore, we will ward against model misspecification by providing evidence that the assumptions made throughout the modeling are well-grounded, since the appropriateness of these assumptions underly whether or not the 'true' model is even under consideration - beyond the type I and type II errors commonly assessed by statistical testing.

We will provide evidence that sub-groups of aspects of the modeling are valid based on conditional independence relationships implied by the aforementioned fundamental modeling assumptions, and lastly provide some general evidence that the complete model sufficiently reconstructs and predicts price traces.

B.3 Independence of dprices and itimes

Recall from Section B.1.4 that *dprices* refers to the differenced price series, and *itimes* refers to the differenced series of times at which prices updated.

From a purely theoretical perspective, the magnitude of price jumps and the length of time said prices persist for are at minimum weakly dependent on one another. Empirically however, the dependence between these two series is very weak over the time horizons typically encountered in the completion of real workloads on

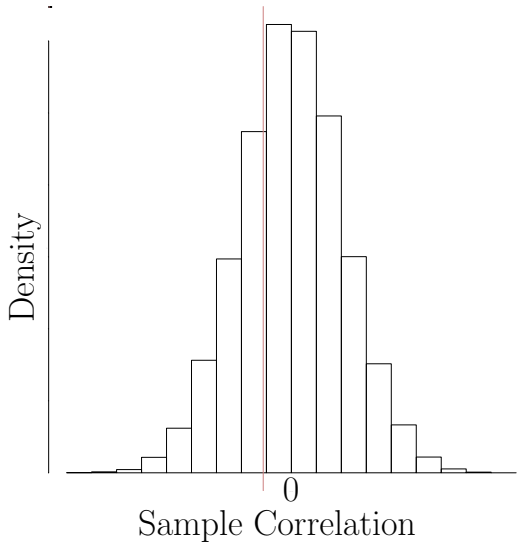

FIGURE B.1: Simulation Experiment of Correlation Between dprices and itimes, demonstrating that the observed correlation is not statistically significant from zero.

an individual basis.

If the two random variables are truly independent, they theoretically have 0 correlation, however in any finite sample some correlation will be observed, with random sign and stochastically decreasing absolute value as sample size increases. The exact distribution of the correlation coefficient based on this sample size of dprice and itime depends on their respective distributions, which are unfortunately unknown. However we can bootstrap from these observations in an attempt to break time dependence between the series, should it exist, and obtain a reference distribution for the correlation coefficient in the flavor of Monte Carlo experiment. Proceeding with this strategy, we obtain the graph found in Figure B.1, which shows that it is not unreasonable that the correlation could truly be zero over this time horizon ($\sim$10 months) with respect to its depicted reference distribution. This implies that it is plausible that dprice and itime could be independent, but further testing would need to be done to truly verify that proposition.

In general proving that two time series are independent can be quite involved

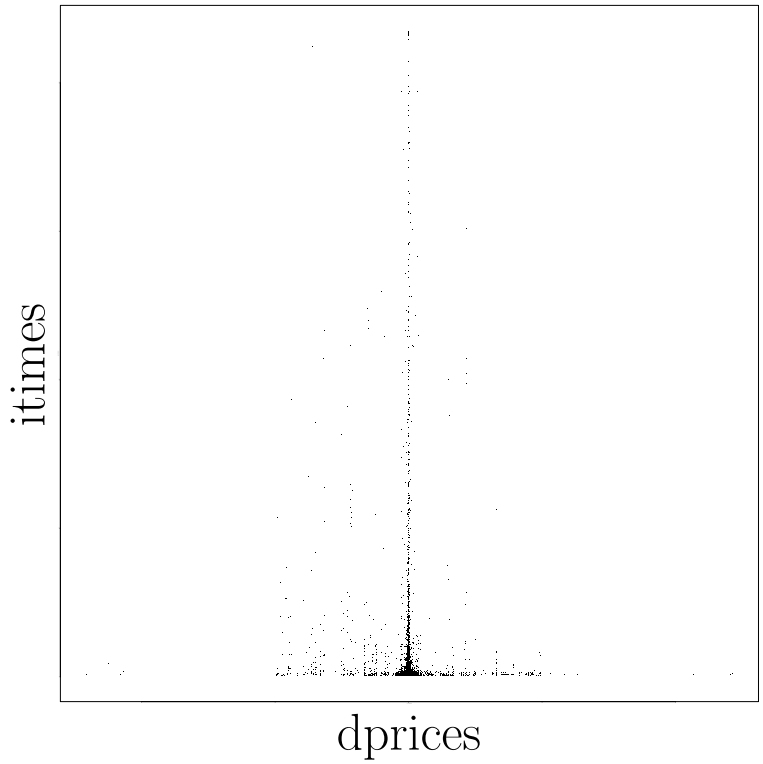

FIGURE B.2: Scatter Plot of dprices Against itimes, demonstrating that there is no obvious non-linear pattern between inter-arrival times and price jumps.

depending intricately upon the nature of both processes. Before getting caught up in these details, we explore the first order relationships graphically. The issue is that dependent random variables can exhibit zero correlation when there is a non-linear relationship between the two variables which averages out to zero linear relationship.

Note that there is not significant asymmetry or otherwise obvious non-linear pattern in the inter-arrival time distribution given the magnitude of the current price jump in Figure B.2, lending weight to the proposition that dependence is weak if it exists. It is true that very large inter-arrival times are only empirically observed for small price magnitudes, however it not clear from this data whether or not this is simply due to the fact that the vast majority of price jumps are small, as are the vast majority of inter-arrival times - that is to say that this observed pattern is also at least partially an artifact of finite sampling. Since many of the points are

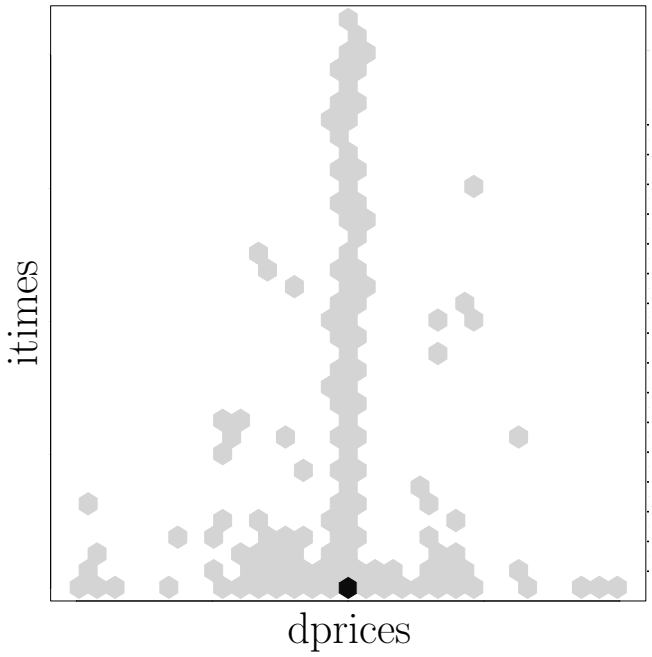

FIGURE B.3: Hexagonal Binning of itimes Against dprices, demonstrating that the vast majority of observations occur with both dprice and itime near zero. It is plausible that the wide range of itimes observed for dprice near 0 is caused by over-sampling and not a fundamental change in its distribution (and vice versa).

overlapping in the above scatter plot, a hexagonally binned version is also included in Figure B.3 to better depict the point density, however the exact values are not observable in the hexagonally binned version.

One might be tempted towards the idea that small price jumps have a different relationship with inter-arrival times than large price jumps, however similar simulation studies have similarly indicated that this is also not the case - or minimally that with this volume of data, the relationship is not very strong.

Further investigation of possible lagged dependencies yields similar findings as is shown in Figure B.4.

What complicates this in the case of time series, is that it is not sufficient to show that the conditional distribution of one series is unaffected by knowledge of the current position of the other series, but also for any combinations of observations

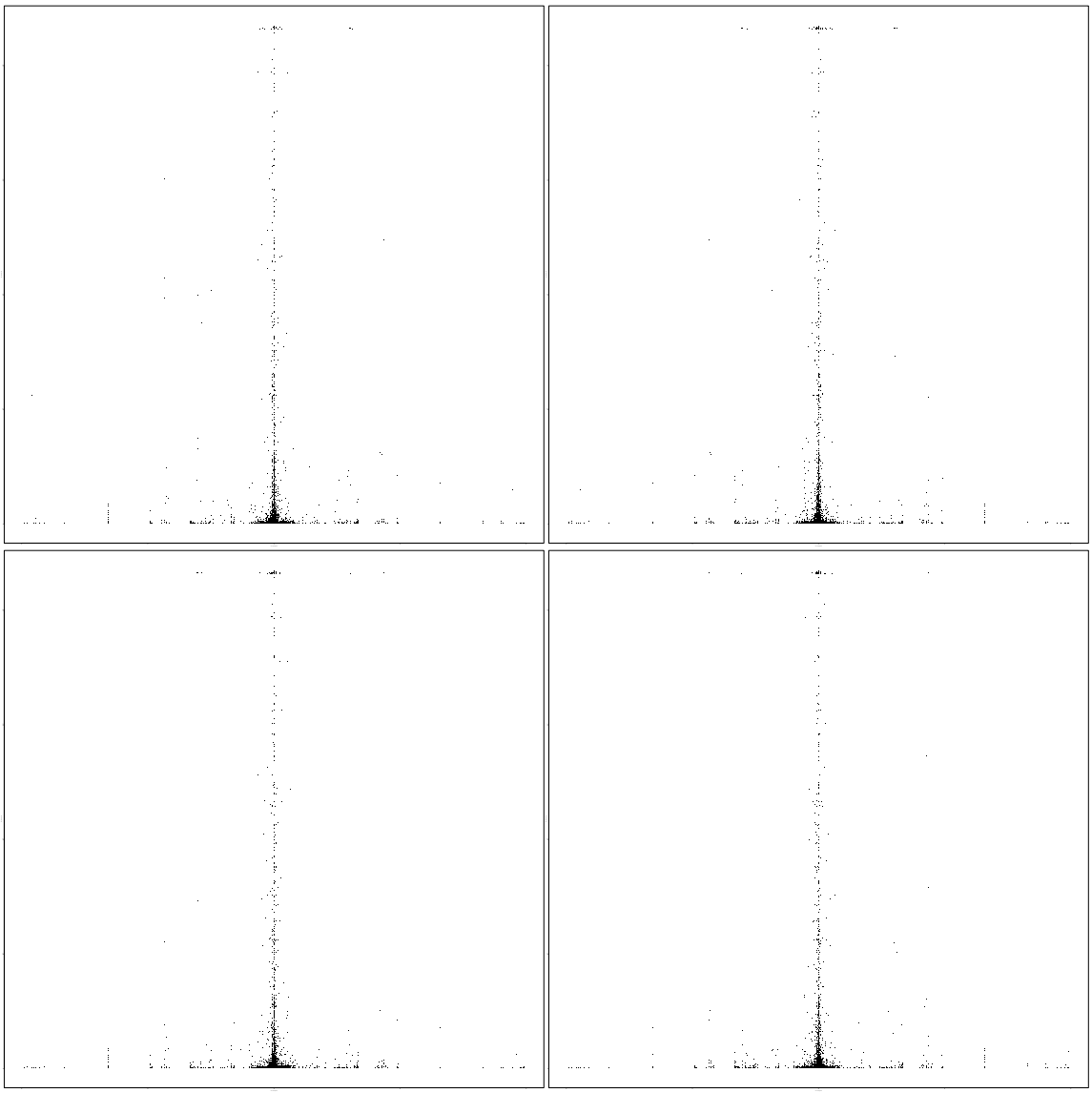

FIGURE B.4: Lagged Scatter Plots of itimes Against dprices

over time. This sheds light on why rigorously testing for independence of time series of unknown form is complicated, and why it ultimately rests on some fundamental assumptions which must be made about the series in question. Fortunately, many of these assumptions are not unreasonable for the market setting, at least over a short time-horizon. For example, while it is known that covariance structures which motivate changes in many pricing markets are dynamic over time, many fixed covariance structure models still see use in finance to great positive economic benefit of their

end users. This is possible for a variety of reasons, but it is not uncommon that the covariance structures simply do not change quickly enough to out-date the utility of the model predictions on the time-scale of financial instrument trading. This is part of the reason why such models based on fixed covariances in finance often need to be re-fit periodically to remain relevant, and also why markets don't simply reach steady states; in essence, the target of the random process is always changing. For further reading on testing for dependence of two streams of time-series data in the case where they can both be modeled as covariance stationary auto-regressive moving average series, see 'Checking the Independence of Two Covariance-Stationary Time Series: A Univariate Resisutal Cross-Correlation Approach,' Larry D. Haugh (*JASA* vol. 71). It is noteworthy to observe that the 'true' model for price jump magnitude is likely more general than the covariance-stationary ARMA models that are assumed within this supplementary reading material.

To summarize,

- Exploratory data analysis and examination of first order dependencies do not reveal any significant trends.
    - The complexity of assumption based rigorous dependence testing for time-series data (especially for the price magnitude series which has extremely interesting and complex self-contained patterns), and the sparsity of paired observations of inter-arrival times and price magnitudes which are mutually far from 0 preclude specifying the nature of dependence, should it exist, in a way that improves the quality of reconstructions and predictions of spot-instance pricing.
- Lack of dependence between inter-arrival times and the magnitude of price jumps does not imply that the prediction problem is hopeless over shot-time

horizons, simply that there is not very strong cross-talk between price updates and time updates.

As mentioned in Sections B.1.3 and B.1.5, there is evidence to suggest that there are interesting patterns which are self-contained and which can be leveraged to obtain realistic price reconstructions and forecasts.

B.4 Differenced Price Series

Since we forecast prices and inter-arrival times independently given the current state of the market and the market history, in this section we focus on the series of sequential magnitudes of the differences between successive prices in abstraction of the length of time they persisted for. Unless otherwise specified the x-axis counts the order of the price jumps, and not the actual time of day etc. throughout this section. We will return to the setting where the x-axis tracks real-time later.

As can be seen in Figure B.5, the series of price jumps is characterized by short bursts of extreme fluctuations, however it is clear that these periods of high volatility do not arrive with a fixed distribution over time, with quasi-cyclic patterns observable. These patterns are related to the actual time elapsed in the spot-price market, and are therefore partially obfuscated in the graph above as the axis is merely tracking the order in which price updates were observed, and not the actual temporal location of the price jump.

Another obvious pattern which is apparent from Figure B.5 is that extreme price spikes tend to occur very close in sequence after one-another. This is also evidenced by a negative lag-1 auto-correlation in the observed series as shown in Figure B.6. This auto-correlation plot also supports the theory of quasi-cyclic patterns, as the values oscillate as they approach 0. The period of this oscillation is not strictly constant, appearing to be on the order of about 17 time-steps or so. Figure B.7

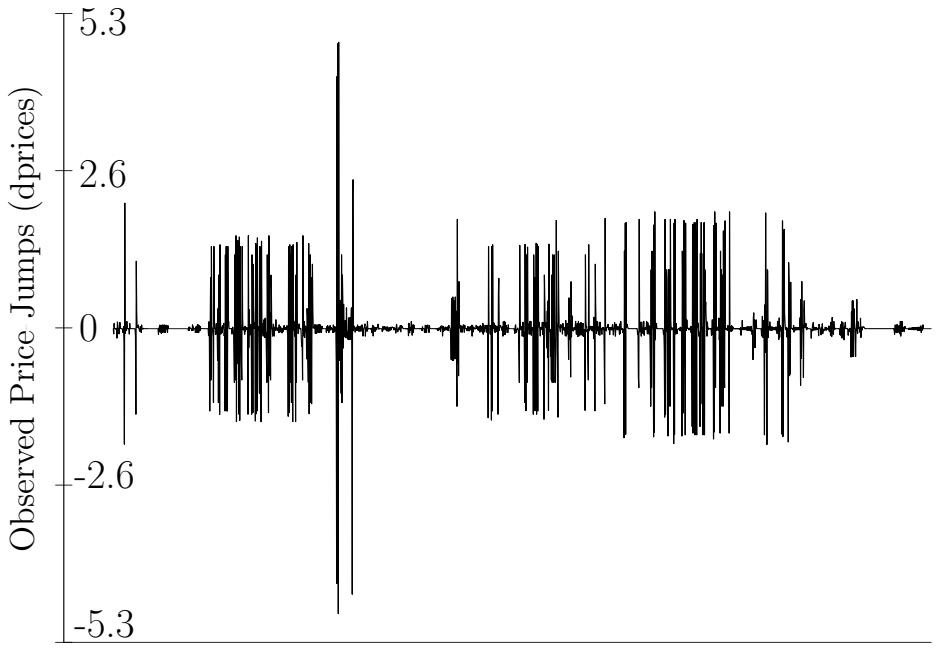

FIGURE B.5: Excerpt of dprices Series. Note that the x-axis only counts the order in which price updates were observed, and not their true temporal location.

demonstrates that this feature of the data is also captured by our proposed method as the simulated series exhibited negative lag-1 auto-correlation with high probability.

The time-marginalized simulations also show a good approximation to the observed time-marginalized distribution, as can be seen in Figure B.8. Minor differences can be identified, especially in the tails; however the histograms are astonishingly similar to each other. Furthermore, the simulations match the observed data series in terms of the time-marginalized summary statistics included in Table B.1. Since not all variables share similar ranges, they are plotted with different vertical ranges in Figure B.9. The important thing to note from these plots, is that the high density region (darker squares) occur at the same locations at the observed statistics (red points). The exception to this is of course the negative minimum, and the maximum price jump which are located at the respective extremes of the simulated minimum and maximum distributions. This is natural since the simulations are based on sig-

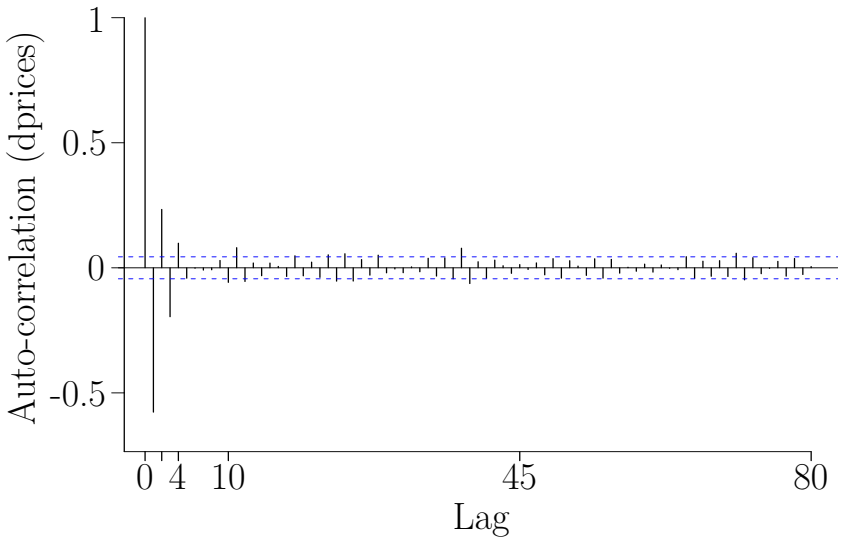

FIGURE B.6: Auto-correlation of the Observed dprices Series. Note the statistically significant negative correlation in successive prices jumps. Another interesting feature is that the correlations oscillate as they approach zero with increasing lag.

nificantly less time than the length of time the training data was collected for.

B.5 Inter-Price Times Series

As with the previous section, unless otherwise specified the x-axis counts the order of the price jumps, and not the actual time of day etc. throughout this section.

The Inter-price times (itimes) series is subject to intense volatility, similar to what we saw in the case of the dprices series. The itimes series however, 'explodes' on a much more regular basis, as can be seen by comparing Figures B.5 and B.11. Another readily observable difference, is that the itimes series is significantly more skewed than the dprices series; this observation is also supported by the observed time-marginalized skewness of 5.33 for the itimes series (compared to only 0.01 observed time-marginalized skewness in the dprices series).

Investigating Figure B.12 we see that successive inter-price times are likely to be positively related, and it is not implausible that there could be long-term pos-

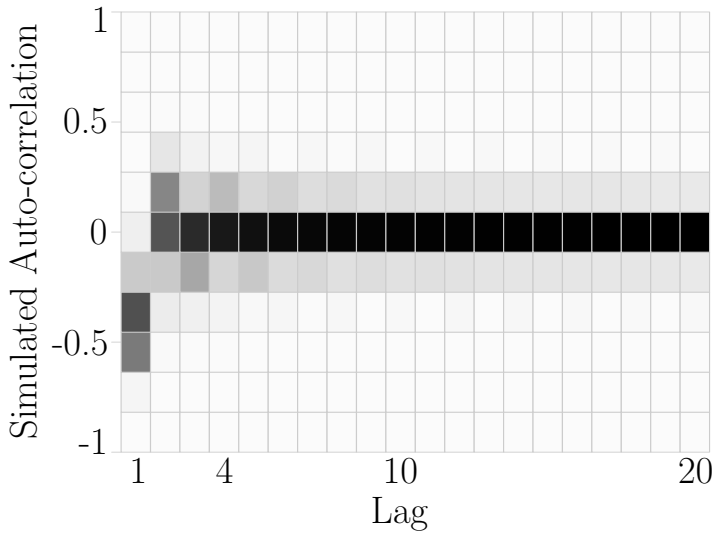

FIGURE B.7: Auto-correlation of Simulated dprices Series. Note that this plot is based on 6000 simulations of a given week, and hence does not necessarily need to match the auto-correlations observed in Figure B.6 exactly. The range $[-1, 1]$ is split into 11 equal pieces and colored according to the frequency at which simulations produced auto-correlations in that range, with black representing high-density and white representing low-density. This demonstrates that our method is able to capture the behavior that large positive price spikes tend to be followed by large negative price spikes, as is demonstrated by the negative first order auto-correlation.

itive linear relationships between inter-arrival times. The positive low-order auto-correlations make intuitive sense, as there are certainly times at which the spot-instance market is busier than others. Short inter-arrival times are likely to be observed during a 'busy' period, for which it is more likely to further observe short inter-arrival times.

Alternatively, this could also be supported by instances in the data where prices fluctuate initially before settling back into a period of relative stability; these could possibly be related to non-instantaneous adjustment of the market to changes in supply on Amazon's side. For an example of how this could occur, consider that consumers may over-estimate the extent of a supply cut, and hence may pay prices they would not consider reasonable under regular supply levels under the perception

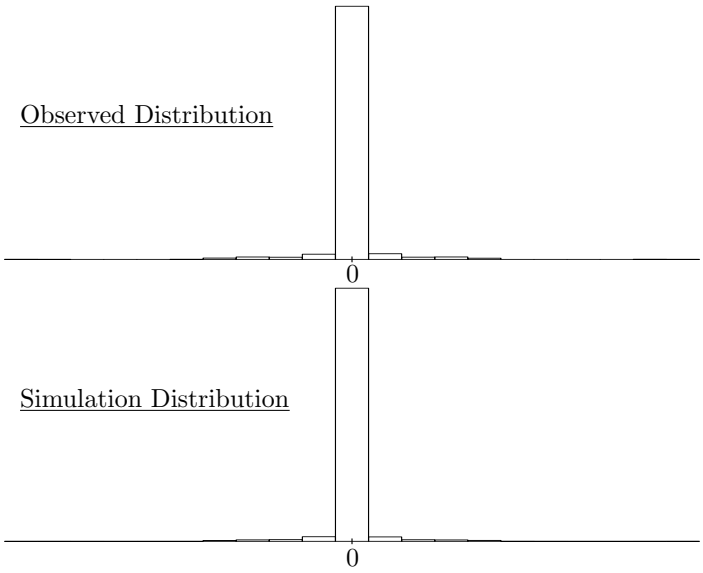

FIGURE B.8: Time-Marginalized Distributions of Observed and Simulated dprices Series. Observe that the shape and location of the histograms match closely with each other, with only minor differences observable in the tail of the distribution.

that supply is suddenly very limited. This is especially plausible when the product is limited-time or otherwise deadline dependent. As time progresses, the true extent of the supply cut is discovered and individuals' valuations adjust to more realistically match their consumption requirements.

These patterns are also captured by the simulated itimes series, as can be seen in Figure B.13. Convergence to 0 is much slower than was the case for the dprices series, with the range of simulated correlations decreasing by about 20% from lag-1 to lag-20. This feature of the simulated series also matches the observed pattern of auto-correlations in Figure B.12, since excluding lag-1 and lag-2, it is not clear that the series truly decreasing from lag-3 to lag-50. The auto-correlations of the simulated series similarly show an initial decrease in correlation up to lag-2, followed by relative stability through lag-20.

The time-marginalized simulations also show a good approximation to the observed time-marginalized distribution, as can be seen in Figure B.14. Furthermore,

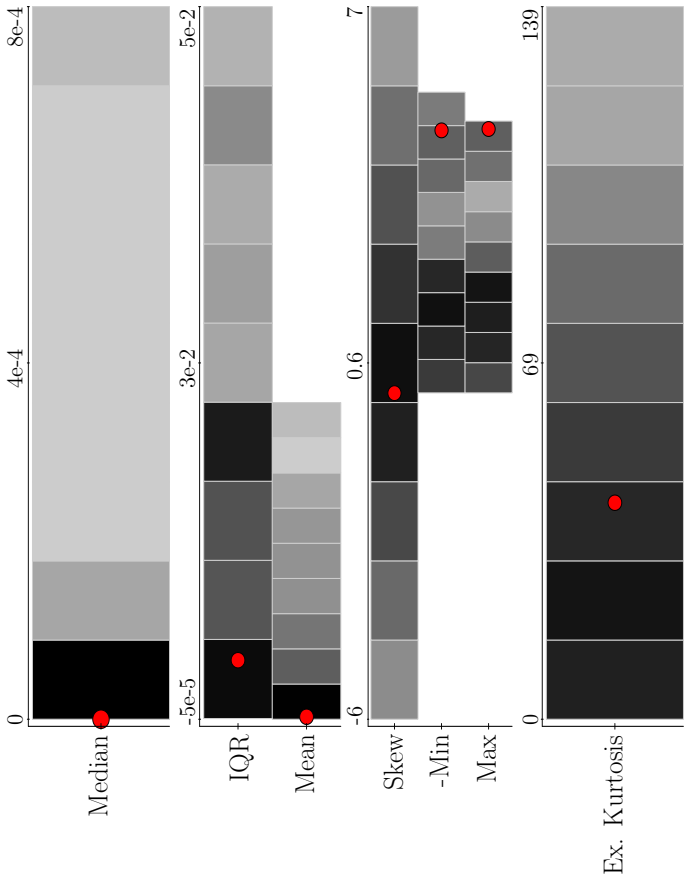

FIGURE B.9: Time-Marginalized Statistics of Observed and Simulated dprices Series. Note that the observed statistics are essentially at the peaks of the distributions based on simulations in nearly all cases, with the obvious exception of the negative minimum and the maximum, which are correctly located near the ends of their distributions.

the simulated series generally does a good job of matching the observed data series in terms of the time-marginalized summary statistics included in Table B.1, as can be seen in Figure B.15. In most columns, the high density region (darker squares) occurs at a similar location as the observed statistics (red points), with the natural exception of the minimum and maximum, which correctly appear at the extremes of their distributions. The median appears to be under estimated slightly, but is still contained within the support of the density of the medians of the simulated series. Furthermore, note that the tail of the median distribution appears to be heavier than

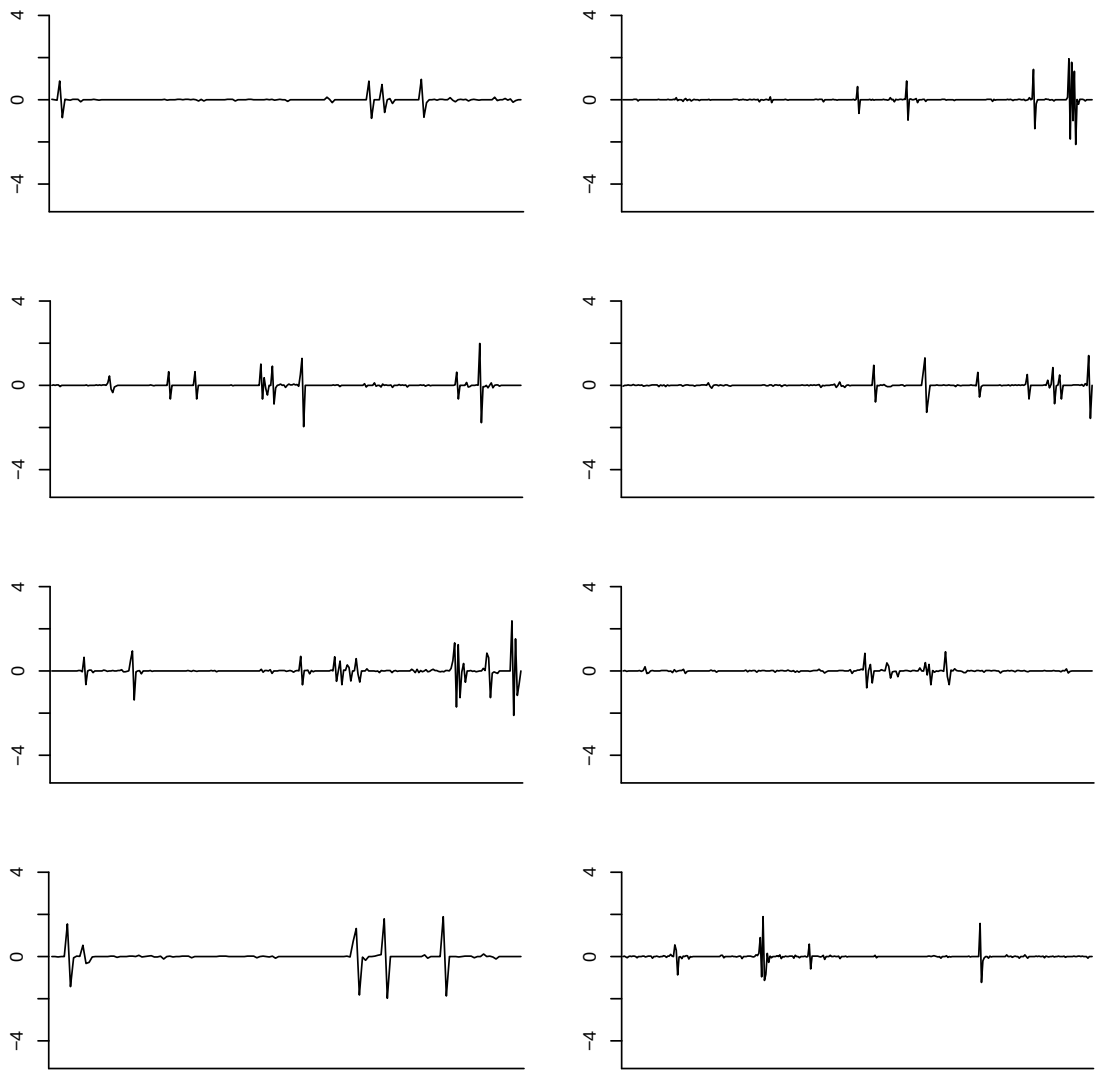

FIGURE B.10: Excerpts of Simulated dprices

most of the others. In fact, it is heavy enough that we cannot be 95% certain that the simulated medians are systematically lower than the observed median.

B.6 Estimating Cost

While it is good that the model is able to match many of the observed properties of the dprices and itimes series, ultimately it is important that the model provides good advice for metrics which form the bottom-line for cloud users. To this effect,

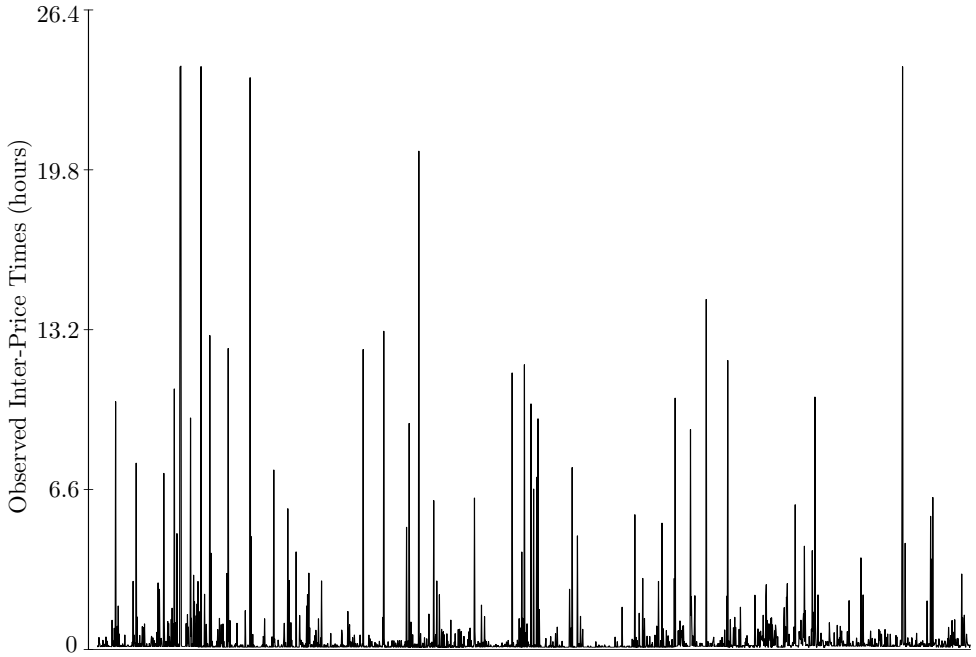

FIGURE B.11: Excerpt of itimes Series. Note that the x-axis only counts the order in which price updates were observed, and not their true temporal location.

see Figure B.16 which shows how the simulated series can predict the real observed cost with high accuracy. This is calculated based on real-time data, unlike the immediately previous sections which analyzed the properties of sequential updates of dprices and itimes without including the real-time information about when those sequential updates actually occurred.

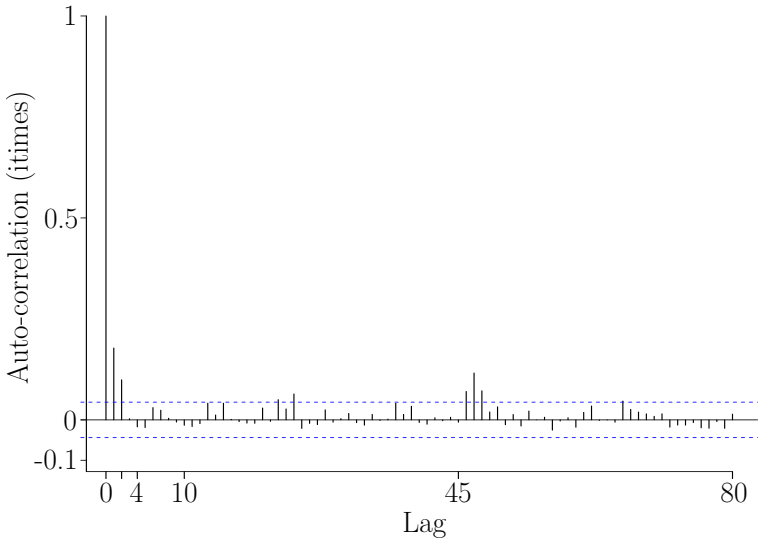

FIGURE B.12: Auto-correlation of the Observed itimes Series. Quasi-cyclic patterns are again visible, with the series oscillating as it approaches zero with increasing lag. As opposed to dprices for which large positive spikes are followed by large negative spikes, inter-price times are positively related with each-other over short time-scales.

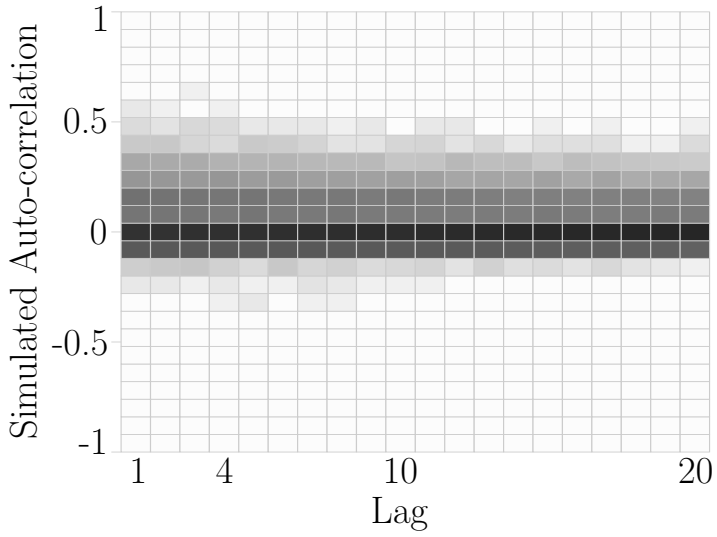

FIGURE B.13: Auto-correlation of Simulated dprices Series. Note that this plot is based on 6000 simulations of a given week, and hence does not necessarily need to match the auto-correlations observed in Figure B.12 exactly. The range $[-1, 1]$ is split into 25 equal pieces and colored according to the frequency at which simulations produced auto-correlations in that range, with black representing high-density and white representing low-density. Observe that low-order lags exhibit positive correlation, and that higher-order lags occasionally show a greater tendency towards positive association as well.

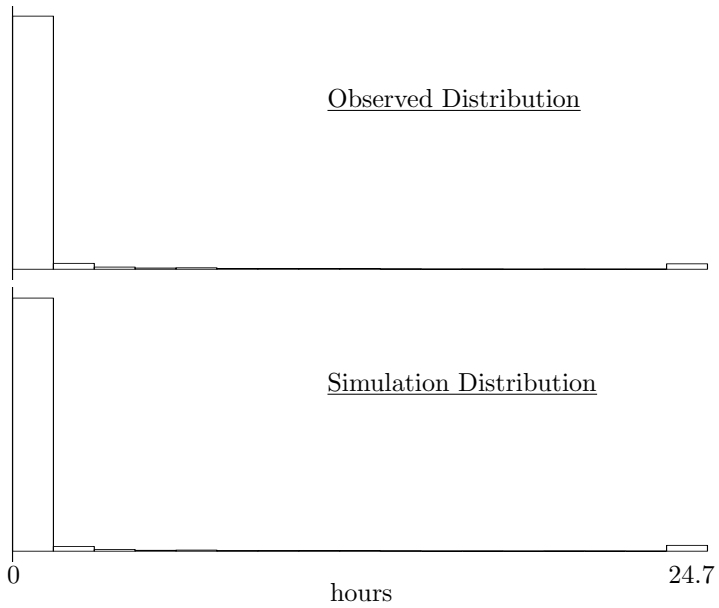

FIGURE B.14: Time-Marginalized Distributions of Observed and Simulated itimes Series. Observe that the shape and location of the histograms match closely with each other, with only minor differences observable in the tail of the distribution. The tails of the histogram are close enough so that whether the observed or simulated distribution appears to have a heavier tail depends on the resolution at which the histograms are viewed.

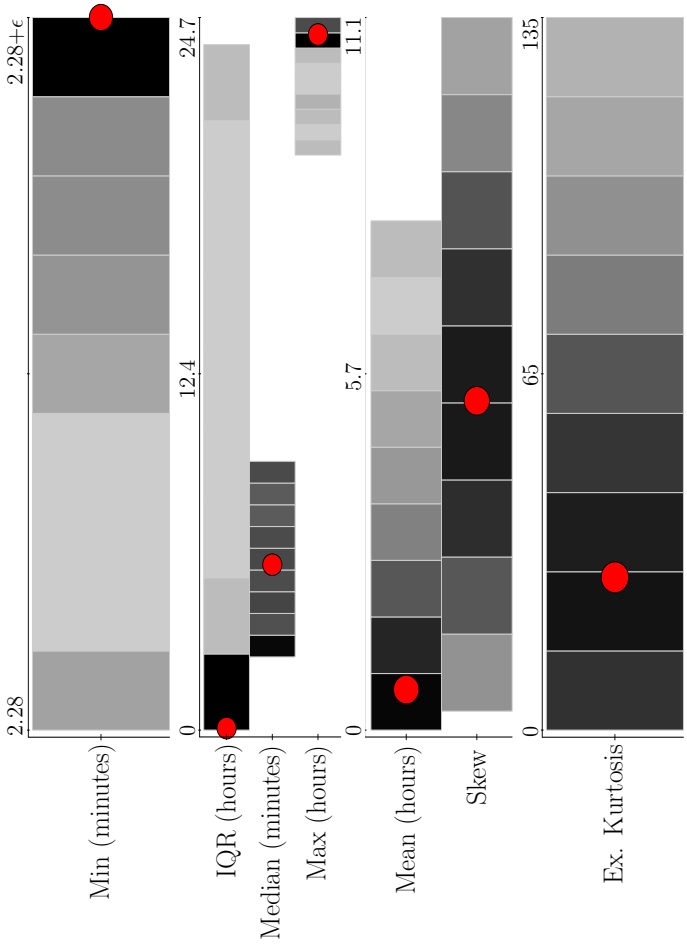

FIGURE B.15: Time-Marginalized Statistics of Observed and Simulated itimes Series. Note that the most of the observed statistics are essentially at the peaks of the distributions based on simulations. The minimum and the maximum, are correctly located near the ends of their distributions; however the Median appears to be under-estimated slightly, though the effect is not statistically significant.

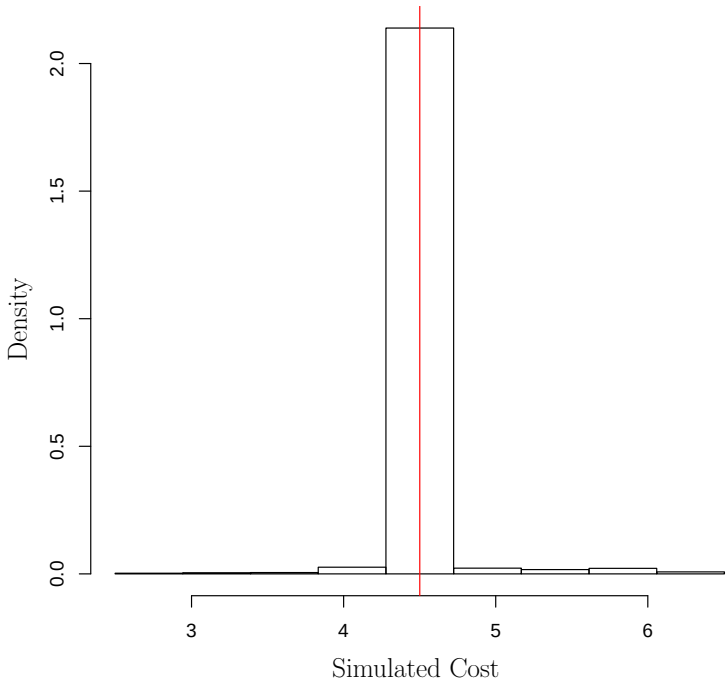

FIGURE B.16: Cost to Obtain Resources as Predicted via Simulated Series vs Actual Observed Cost.

Bibliography

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] H. Amur, J. Cipar, V. Gupta, G. R. Ganger, M. A. Kozuch, and K. Schwan. Robust and flexible power-proportional storage. In *Proceedings of the 2010 ACM Symposium on Cloud Computing*, Indianapolis, Indiana, USA, June 2010.

[3] 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*, Miami, Florida, USA, Aug. 2010.

[4] 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.

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

[6] P. G. Brown. Overview of SciDB: Large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 963–968, Indianapolis, Indiana, USA, June 2010.

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

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

In *Proceedings of the 2011 ACM/IEEE Supercomputing Conference*, Seattle, Washington, USA, Nov. 2011.

[9] N. Chohan, C. Castillo, M. Spreitzer, M. Steinder, A. 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.

[10] 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.

[11] 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.

[12] 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.

[13] A. B. Downey. *A model for speedup of parallel programs*. University of California, Berkeley, Computer Science Division, 1997.

[14] 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.

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

[16] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proceedings of the 2005 USENIX Symposium on Networked Systems Design and Implementation*, Boston, Massachusetts, USA, May 2005.

[17] 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.

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

[19] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics. In *Proceedings of the 2011 ACM Symposium on Cloud Computing*, Cascais, Portugal, Oct. 2011.

[20] T. Hofmann. Probabilistic latent semantic indexing. In *Proceedings of the 1999 ACM SIGIR International Conference on Research and Development in Information Retrieval*, pages 50–57, Berkeley, California, USA, Aug. 1999.

[21] 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.

[22] B. Huang, M. Boehm, Y. Tian, B. Reinwald, S. Tatikonda, and F. R. Reiss. Resource elasticity for large-scale machine learning. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, Melbourne, Australia, May 2015.

[23] B. Huang, N. W. Jarrett, S. Babu, S. Mukherjee, and J. Yang. Cumulon: Cloud-based statistical analysis from users perspective. *IEEE Data Engineering Bulletin*, 37(3):77–89, Sept. 2014.

[24] 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`.

[25] B. Huang, N. W. D. Jarrett, S. Babu, S. Mukherjee, and J. Yang. Cümülön: Matrix-based data analytics in the cloud with spot instances. *Proceedings of the VLDB Endowment*, 9(3):156–167, 2015.

[26] 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 *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*, pages 228–235. IEEE, 2013.

[27] 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.

[28] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A peta-scale graph mining system. In *Proceedings of the 2009 International Conference on Data Mining*, pages 229–238, Miami, Florida, USA, Dec. 2009.

[29] S. Khatua and N. Mukherjee. Application-centric resource provisioning for amazon ec2 spot instances. In *Euro-Par 2013 Parallel Processing*, pages 267–278. Springer, 2013.

[30] D. D. Lee and H. S. Seung. Algorithms for non-negative matrix factorization. In *Proceedings of the 2000 Advances in Neural Information Processing Systems*, pages 556–562, Denver, Colorado, USA, Dec. 2000.

[31] 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.

[32] 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.

[33] M. Mazzucco and M. Dumas. Achieving performance and availability guarantees with spot instances. In *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*, pages 296–303. IEEE, 2011.

[34] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 1099–1110, Vancouver, Canada, June 2008.

[35] 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.

[36] 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.

[37] 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*, Orlando, Florida, USA, Mar. 2012.

[38] M. Taifi, J. Y. Shi, and A. Khreishah. Spotmpi: A framework for auction-based hpc computing using amazon spot instances. In *Algorithms and Architectures for Parallel Processing*, pages 109–120. Springer, 2011.

[39] 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*, Honolulu, Hawaii, USA, June 2012.

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

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

[42] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[43] 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.

[44] W. Voorsluys and R. Buyya. Reliable provisioning of spot instances for compute-intensive applications. In *Advanced Information Networking and Applications (AINA), 2012 IEEE 26th International Conference on*, pages 542–549. IEEE, 2012.

[45] P. Wang, Y. Qi, D. Hui, L. Rao, and X. Liu. Present or future: Optimal pricing for spot instances. In *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*, pages 410–419. IEEE, 2013.

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

[47] M. Zafer, Y. Song, and K.-W. Lee. Optimal bids for spot vms in a cloud for deadline constrained jobs. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 75–82. IEEE, 2012.

[48] 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 *Proceedings of the 2012 USENIX Symposium on Networked Systems Design and Implementation*, pages 15–28, San Jose, California, USA, Apr. 2012.

[49] Q. Zhang, Q. Zhu, and R. Boutaba. Dynamic resource allocation for spot markets in cloud computing environments. In *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*, pages 178–185. IEEE, 2011.

[50] 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.

[51] 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.

Biography

Botong Huang was born on December 25th, 1989 in Nanchang, Jiangxi, China. In May 2010, he received his bachelor's degree from Peking University, Beijing, China, majoring in Computer Science and minoring in Economics. He received his master's degree in May 2013 and the PhD degree in Feburary 2016 from Duke University.

At Duke, Botong did his research on cloud computing under the supervision of Professor Jun Yang and Shivnath Babu. Three major papers [21, 23, 25] are published, which corresponds to Chapter 2 and 3 of this thesis. [24] is the full version of [25] as a technical report. The content in Chapter 4 is currently under submission.

In the summer of 2014, Botong did a reserach internship in the SystemML team at IBM Research Alamden. He worked on the resource optimization aspect of SystemML running on top of Yarn. This work is later published in [22].