Finding the Right Cloud Configuration for Analytics Clusters

Muhammad Bilal*
UCLouvain and
IST(ULisboa)/INESC-ID

Marco Canini
KAUST

Rodrigo Rodrigues
IST(ULisboa)/INESC-ID

ABSTRACT

Finding good cloud configurations for deploying a single distributed system is already a challenging task, and it becomes substantially harder when a data analytics cluster is formed by multiple distributed systems since the search space becomes exponentially larger. In particular, recent proposals for single system deployments rely on benchmarking runs that become prohibitively expensive as we shift to joint optimization of multiple systems, as users have to wait until the end of a long optimization run to start the production run of their job.

We propose Vanir, an optimization framework designed to operate in an ecosystem of multiple distributed systems forming an analytics cluster. To deal with this large search space, Vanir takes the approach of quickly finding a *good enough* configuration and then attempts to further optimize the configuration during production runs. This is achieved by combining a series of techniques in a novel way, namely a metrics-based optimizer for the benchmarking runs, and a Mondrian forest-based performance model and transfer learning during production runs. Our results show that Vanir can find deployments that perform comparably to the ones found by state-of-the-art single-system cloud configuration optimizers while spending 2× fewer benchmarking runs. This leads to an overall search cost that is 1.3-24× lower compared to the state-of-the-art. Additionally, when transfer learning can be used, Vanir can minimize the benchmarking runs even further, and use online optimization to achieve a performance comparable to the deployments found by today's single-system frameworks.

*Work done in part while author was interning at KAUST.

SoCC '20, October 19–21, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8137-6/20/10…$15.00
https://doi.org/10.1145/3419111.3421305

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; • **Social and professional topics → Management of computing and information systems**.

ACM Reference Format:
Muhammad Bilal, Marco Canini, and Rodrigo Rodrigues. 2020. Finding the Right Cloud Configuration for Analytics Clusters. In *ACM Symposium on Cloud Computing (SoCC '20), October 19–21, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3419111.3421305

1 INTRODUCTION

Data analytics is part of the computational ecosystem of virtually every organization that seeks to extract value from the data it collects. In such computational infrastructures, a wide variety of use cases *do not* execute on a standalone framework — instead, as depicted in Figure 1, they execute in an ecosystem of multiple frameworks that connect in a graph-like structure to form an analytics cluster. The pipeline depicted in this figure is very similar to the real-world deployments for Periodic ETL [3], or Lambda architectures for aggregating clickstream events [11].

A common way to deploy these analytics clusters is to *acquire resources* in a cloud environment and do so *on demand*, whenever (typically recurring) jobs need to execute. However, when executing jobs in the cloud, ensuring that the *right resources* are allocated is paramount, not only due to cost efficiency but also to satisfy strict service-level objectives (SLOs) on job completion times. Thus, deploying analytics clusters in the cloud requires solving a *cloud configuration problem*: for each framework, determine (1) how many instances to use, and (2) which type of instances to use? Choosing the *right configuration* is crucial because, when such decisions are wrong or sub-optimal, jobs fail to meet their deadlines and/or execution costs increase several-fold, as we show in §2.2.

Prior methods for selecting cloud configurations [13, 17, 27–29, 42] do not consider the existence of multiple frameworks in an analytics cluster, but instead focus on configuring a single framework, such as Spark or Hadoop. Furthermore, all possible ways of applying these prior methods in the multi-framework analytics cluster setting have significant

Figure 1: Example data analytics cluster.

limitations. The *first way* is to use these methods to optimize each framework individually. However, as our results show, the coupling between different frameworks and incorrect distribution of budget causes this approach to miss opportunities to improve performance. The *second way* is to consider the entire analytics cluster as if it were a "single-large-system," simultaneously modifying the configuration of all frameworks. This approach enables joint optimization but increases the size of the configuration search space exponentially with the number of frameworks. Exploring a larger space naturally takes longer, causing a costly delay before users can deploy their clusters in production.

We present **Vanir**, a system for finding the right cloud configuration for multi-framework data analytics clusters. Vanir is designed for a setting where a user needs to provision and set up an on-demand analytics cluster for each run of a batch processing job. In this scenario, it is often the case that a large fraction of these deployments are recurring, as supported by reports that more than 40% of the jobs in production clusters are recurring computations [12, 24, 39]. After the job executes, the cluster is terminated or scaled down to avoid any further costs [5]. The main principle that Vanir adopts to cope with a large configuration search space is to find a *good enough* configuration via a fast benchmarking phase, and optimize that configuration during production runs, as the job recurs. A *good enough* configuration is one that satisfies user-defined SLO constraints on *both* cost and execution time. The design of Vanir combines in a novel way several techniques needed to jointly optimize the resource requirements for multi-framework analytics jobs. In particular, during a preliminary benchmarking phase, Vanir uses a metrics-based optimizer to quickly determine an initial configuration. In this stage, Vanir also attempts, whenever applicable, to use transfer learning and similarity between jobs to reuse knowledge from previous jobs. Then, to tackle the inherent limitations of a quick benchmarking search within a large search space, Vanir fine-tunes the configuration with the help of a performance model that is updated continuously with each successive run of a recurrent job. To select new configurations, Vanir employs Mondrian forests [32], an online random forest approach, aimed at incrementally improving the configuration.

We implement a Vanir prototype (which we plan to release as open-source) and use it to guide the deployment of nine different jobs on two different pipelines (including the one in Figure 1) using AWS EC2. We evaluate Vanir against two baselines corresponding to the above-mentioned ways of using existing optimization methods to handle multi-framework analytics clusters. Our results show that Vanir finds configurations that are comparable to completely offline methods. More importantly, this is achieved while requiring only half of the benchmarking runs, and with a total cost of optimization that is 1.3-24× lower than state-of-the-art methods.

2 ANALYTICS CLUSTERS CONFIG

Analytics clusters are widely prevalent in modern data-driven organizations. A rich ecosystem of software systems gives organizations the freedom to create sophisticated analytics clusters composed of a multitude of frameworks such as Hadoop, Spark, Cassandra, or Flink. We begin by defining the cloud configuration problem and then motivate our work by demonstrating the importance of selecting good cloud configurations. We further outline the challenges and discuss baseline solutions.

2.1 The cloud configuration problem

We formalize the cloud configuration problem as the problem of finding a configuration of resources that minimizes job execution time. A job is a combination of application (for example a random forest application in Spark) and the input data. Since a cluster comprises of multiple frameworks, we want to *jointly identify both the type and number of instances for each framework* within the cluster. Formally, a cloud configuration is denoted as a vector $C = \{\langle N_1, I_1\rangle, \ldots, \langle N_n, I_n\rangle\}$ where N_F is the number of instances for framework $F \in \{1, \ldots, n\}$ and I_F is the corresponding instance type.

As a validity condition, we require that any *valid cloud configuration satisfies user-specified constraints on the maximum execution time and maximum execution cost.* In addition, we consider that the number of instances of each framework is chosen from a finite set of possible values. The minimum and maximum values across all frameworks yield the *search space bounds.*

An instance type defines the CPU, memory, and storage capabilities of an instance. Popular cloud providers group available instance types based on their instance family (e.g., general-purpose, compute-optimized) and instance size (e.g., large, xlarge). The instance family expresses the class of hardware specifications that may best meet the requirements of different applications as well as a CPU-memory ratio. The instance size, in turn, allows for choosing the number of virtual CPUs, amounts of memory, etc.

2.2 Selecting good configurations matters

We illustrate the value of picking a good cloud configuration by reporting the ratio of maximum to minimum execution time and execution cost for all the configurations that were

Job	Max-Min Ratio	
	Time	Cost
Logistic Regression (lr)	2.7×	2.4×
Random Forests (rf)	5.7×	8.3×
PageRank (pr)	3.4×	3.6×
Gradient Boosted Trees (gbt)	8.0×	77.1×
Nweight (nw)	3.5×	5.0×
Shortest Paths (sp)	5.2×	4.3×
Connected Components (cc)	2.3×	9.8×
Label Prop. Algorithm (lpa)	2.0×	2.8×
Price Predictor (pp)	2.9×	2.7×

Table 1: Max to Min ratio for execution time and execution cost of the tested configurations.

used in our evaluation of several optimization methods (§ 7) and for the 9 jobs mentioned in § 7.1. Table 1 shows that selecting a good cloud configuration can drastically decrease job execution time. In particular, the max to min ratio for execution time and execution cost is roughly 2-8× and 2.4-77×, respectively. Thus, optimizing cloud configuration can be crucial to satisfying time and/or monetary SLOs. Note that we do not show configurations where jobs take longer than our maximum time constraint (2,400s). Additionally, we omit results from several configurations that lead to job failures (mainly due to insufficient memory resources). As such, the ratios in Table 1 are a conservative report.

2.3 Challenges

While there are a few recent solutions for finding an appropriate configuration for deploying a single framework in the cloud [13, 27–29, 42], moving to multi-framework analytics clusters raises several additional challenges.

Intrinsic performance coupling. The cloud configuration problem does not decompose into several isolated, per-framework optimizations. This is because the *end-to-end* system performance depends on the performance of each framework, which is coupled with the performance of other interacting frameworks. Moreover, performance coupling means that one cannot simply optimize the cluster configuration one framework at a time following a natural, pre-established sequential order. The baseline method (Baseline 1) described below highlights that not accounting for performance coupling leads to sub-optimal configurations (§7.2).

Huge configuration space. Jointly optimizing multiple frameworks inevitably leads to a large configuration space, which increases exponentially with every additional framework. Thus the search for possible configurations has inherent scalability problems in this new setting. As such, traditional solutions that use exclusively offline optimization are not practical with a large configuration space. Our results in §7.2 show that the search time and search cost of an exclusively offline method can be high, and it may not be practical to use them.

Uninformative offline profiling. Gathering profiling information offline is a typical strategy to enhance the configuration search speed [18, 19, 31, 43]. However, in these methods, it is necessary to profile information for *all possible* configurations, which becomes untenable in our setting, since the number of configurations grows exponentially with the number of frameworks. We show that, in our solution, a short benchmarking phase (combined with transfer learning where possible) eliminates the need for comprehensive offline profiling.

2.4 Baseline solutions

As we outlined, there are two ways of adapting the state-of-the-art solutions to work with multi-framework analytics clusters. We will use the following two solutions as baselines to compare against our method.

Baseline 1. Baseline 1 consists of applying a traditional black-box algorithm like Bayesian optimization with Gaussian processes (used in Cherrypick [13]), to optimize one framework at a time. The main issue is that the order in which each framework is optimized influences the optimization outcome. Since the best order depends on the application and the cluster composition under consideration, we follow the DAG of the analytics cluster as the default order. Note that, even though the baseline optimizes one framework at-a-time, they are not optimized in isolation. In other words, to optimize a given framework, we execute the job in a multi-framework setting, so that the optimization of that single framework is done in its actual execution environment. At a high level, the main limitation of Baseline 1 is that it has a limited view of the configuration search space since it determines the best configuration for only one framework at a time in the specified sequence. Thus, it only observes a smaller subset of the full configuration space and might be unable to explore better configurations.

Baseline 2. For Baseline 2, we also use Bayesian optimization with Gaussian processes, except that this baseline treats the whole cluster as a single black-box system. Thus it configures all of the frameworks simultaneously. Unlike Baseline 1, Baseline 2 is capable of exploring the entire configuration search space.

Baseline 1 and 2 are *offline* methods that require all optimization runs to be executed on a representative input dataset before the job can be deployed in production.

3 OVERVIEW OF VANIR

Vanir solves the cloud configuration problem by splitting the optimization process into two phases. **Phase 1:** A quick

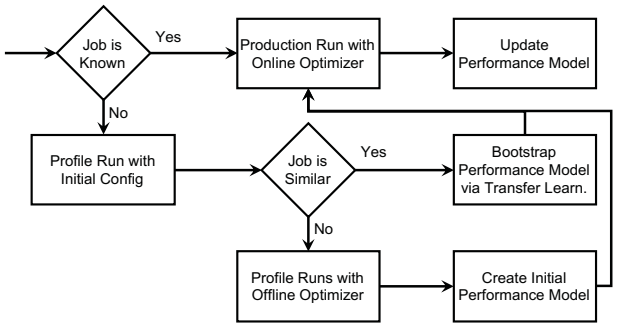

Figure 2: Configuration optimization workflow in Vanir.

heuristics-based benchmarking phase that determines a *good enough* initial configuration. **Phase 2:** A production optimization phase that uses machine learning to progressively find a better configuration. To further improve the performance and scalability of the solution, Vanir adopts a similarity scoring mechanism to bypass the benchmarking phase, if possible.

The two phases use separate optimizer components. The benchmarking phase uses the offline optimizer (whose runs are utilized for benchmarking purposes). The production optimization phase uses the online optimizer (wherein each run is an actual production run that is used to incrementally update the performance model for improving the configuration for the next job iteration). The online optimization process concludes after a user-specified number of runs.

We now discuss Vanir's usage for different high-level workflow scenarios. The next sections present each component in detail and describe our design rationale.

Workflow. Figure 2 overviews Vanir's optimization process. Whenever a job request arrives, Vanir distinguishes between three scenarios:

Known: The job is the same as a previously executed job.

Similar: The job is new (i.e., not seen before), yet it is similar to a previously executed job.

Unknown: The job is new and not sufficiently similar to any previously executed job.

Vanir handles each of these scenarios as follows:

Known. The request for another run of a recurring job is passed directly to the online optimizer. The online optimizer uses the performance model for this job to select a new cloud configuration to test. Once a production run is concluded, Vanir updates the performance model for future optimization.

Similar. When a new job arrives, Vanir executes it on an initial profiling configuration and scores it based on similarity with previously executed jobs. If the job has a sufficient degree of similarity to one of the existing jobs, the job is passed to the online optimizer. The online optimizer loads the performance model for the most similar job and uses transfer learning to bootstrap a performance model for the new job. This performance model is then used as an initial state for the online optimizer to search for an appropriate configuration of the new job.

Unknown. When a new job does not satisfy the similarity filter, the job request is passed to the offline optimizer. The optimizer finds a good enough configuration that meets the user constraints and then launches a production run using that configuration. Subsequent submissions of the same job are optimized using the online optimizer.

4 OFFLINE OPTIMIZER

The goal of the offline optimizer is to quickly find a *good enough* configuration that meets user-provided performance constraints, even if this comes at the expense of the quality of the chosen configuration. Owing to incremental improvements that the online optimizer will apply for subsequent job submissions, this is a reasonable trade-off to scale to large configuration search spaces.

Our design uses a metrics-based algorithm as the offline optimizer, which uses CPU and memory resource utilization metrics (monitored during profiling runs) to determine the configuration of each framework. Note that our aim in this part of the design is to illustrate the advantages of decomposing the configuration problem into an offline and online phase; one could replace our offline optimizer with an alternate one or even a static configuration based on domain knowledge.

The algorithm proceeds iteratively and consists of three phases: *Init*, *Resource Increase* and *Resource Adjustment*. Before we dive into the details of these phases, let us first develop the intuition behind our offline algorithm. Our first goal is to find a good enough configuration that meets the user-specified constraints. To achieve this, we start by rapidly increasing the resource allocation so that we swiftly reach a reasonable set of valid configurations (i.e., that satisfy both execution time and cost constraints), allowing the algorithm to choose the one with the best execution time. However, if this phase only finds one valid configuration or none,[1] then, at a finer granularity, the algorithm reduces the execution cost by decreasing the resources allocated to frameworks with low resource utilization.

This simple algorithm does not deal with different instance families; it sticks with a general-purpose instance and modifies instance size and number of instances. This is intended to make the benchmarking phase short. The online optimizer handles different instance types.

[1] A user could specify execution cost or execution time constraints that are too stringent for the given search space. If the offline optimizer fails to find any valid configuration, then the constraints should be revised.

4.1 Metrics-based algorithm

Init Phase: The optimizer first performs a profiling run on a pre-specified configuration to obtain an initial performance sample. We use an initial configuration with 4 instances of instance type $m5.large$ for each framework.

Resource Increase Phase: Then, in a sequence of profiling runs, the number of instances N_F allocated to each framework F increases according to monitored CPU and memory utilization as follows:

$$N_F = \begin{cases} 2 \cdot N_F & cpu_F + mem_F > \mu_1 \\ s + N_F & \text{otherwise} \end{cases}$$

That is, N_F doubles as long as the mean utilization of CPU (cpu_F) and memory (mem_F) cumulatively remain above a threshold μ_1. Otherwise, a lower sum of CPU and memory utilization indicates that doubling resources would be an over-allocation. In this case, N_F increases by a constant s with each profiling run. We use $s = 2$ and we discuss how to set the thresholds in §4.2. Since the search space bounds impose a limit on the number of instances of a particular size, if N_F reaches this limit, the algorithm shifts to the next larger instance size.

The resource increase phase applies to all the frameworks at once until one of two conditions occurs: (1) the execution time of the current configuration worsens as compared to the execution time of the previous configuration, or (2) the previous configuration was valid but the current configuration is not.

When either of these two conditions occurs, the optimizer checks whether it found more than one valid configuration. If that is the case, it terminates by returning the configuration with the best execution time. Otherwise, the optimizer enters a fine-grained resource adjustment phase to find a better configuration.

Resource Adjustment Phase: The optimizer seeks to find valid configurations by decreasing resources to decrease the overall execution cost. During another series of profiling runs, any framework whose CPU and memory utilization are not above certain thresholds sees its resources decreased. This terminates when the current configuration is not valid while the previous one was. The adjustment in the number of instances is done as:

$$N_F = \begin{cases} N_F/2 & (cpu_F < \mu_2) \wedge (mem_F < \mu_3) \\ N_F & (cpu_F \geq \mu_2) \wedge (mem_F \geq \mu_3) \\ N_F - s & \text{otherwise} \end{cases}$$

The rationale behind the way we change N_F is that we want to have the ability to make significant adjustments (halving or doubling N_F) when the utilization metrics are either too high or too low. Otherwise, we enable fine-grained adjustments (increments or decrements to N_F) to avoid over- or under-allocating the resources by a large margin.

4.2 Tuning the offline optimizer

The resource increase and adjustment phases depend on 3 thresholds that we set empirically. μ_1 controls the rate at which resources are increased w.r.t. resource utilization. A value of 100 means that the combined utilization of CPU and memory is 100%. The maximum value of this parameter is 200. Setting the value close to 200 makes the offline optimizer more conservative in increasing resources and likely slower. Conversely, setting it closer to 0 makes the optimizer more aggressive, which is likely to double the resources at every benchmarking run. μ_2 and μ_3 control the aggressiveness with which the resource adjustment phase decreases resources. The possible values range from 0 to 100. A low value for these parameters means fewer resources taken back from the frameworks and vice versa. In our experiments, we use the following threshold values: $\mu_1 = 100$, $\mu_2 = 50$ and $\mu_3 = 50$. These values strike a balance between aggressiveness and over-allocation of the resources in the different phases of the offline optimizer. A full analysis of these thresholds is outside the scope of this paper.

5 ONLINE OPTIMIZER

The online optimizer aims to further refine the configuration during recurring production runs of the same job. There are three key capabilities that an online optimizer should have: (1) a method to model the job performance on different configurations, (2) the ability to update the performance model at the end of each production run, and (3) a way to select the next configuration (i.e., an *acquisition method*) that potentially improves performance.

In Vanir, the online optimizer maintains and uses a job-specific performance model $M\colon C \rightarrow t$ to predict the job execution time t of any given valid configuration C. Due to the challenges described in §2.3, generally, M is non-convex and its derivative is not available. This precludes standard optimization methods for finding the best cloud configuration. Moreover, M is not necessarily accurate, which requires exploratory production runs to make the model more accurate. To tackle these challenges, we will use Mondrian forests as a fundamental building block.

5.1 Mondrian forests

Mondrian forests use Mondrian processes [36] to construct ensembles of random decision trees. They can be trained in batch or online mode. We use MFs because of three important features [32]. First, MFs gracefully handle predictions on data points outside the space seen in the training data. In the absence of an extensive training dataset, as in our case, this property is of particular importance. Second, online training of MFs has higher accuracy than online RFs for the same amount of training data. Third, online MFs can provide

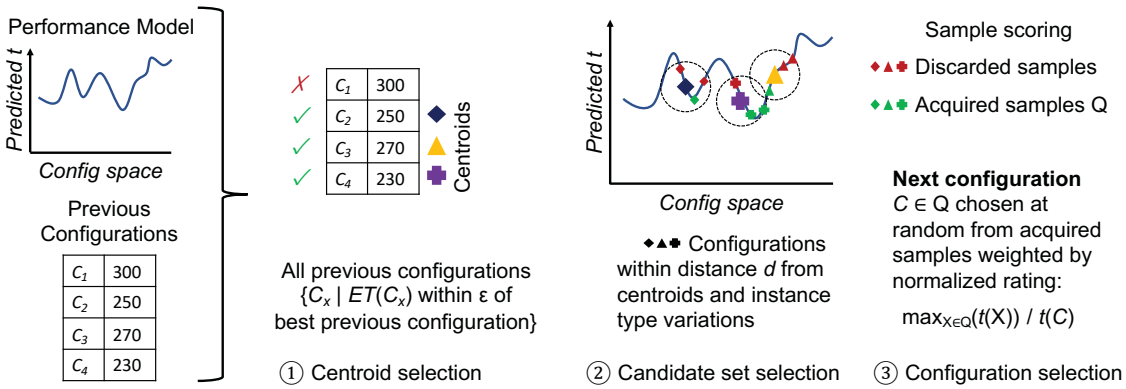

Figure 3: Acquisition method used by the online optimizer.

prediction accuracy comparable to batch-trained RFs. These features make MFs more suitable for our case than the online versions of RFs, such as the one presented in [20, 37].

In Vanir, we use MFs as regressors. Configuration C is the vector of input features, and the output prediction is the job execution time t. Each production run of a configuration is used to update the model. We train the initial MF model M for a job using the data available from the offline optimization phase. Except when a new unknown job has high similarity to other jobs, where we use transfer learning for bootstrapping the initial model of similar jobs (c.f. §6).

5.2 Acquisition method

The acquisition method picks a new test configuration that potentially improves job performance and helps refine the model's accuracy. However, as discussed, the performance model is built incrementally and may be inaccurate (especially in the early stages of a recurring job). Also, due to the large search space, a brute force evaluation of the performance model is inefficient. Therefore, our acquisition method evaluates the performance model on a narrowed down candidate set of configurations before selecting the next configuration for a production run.

We start by providing an intuitive idea of how the algorithm works, as illustrated in Figure 3. At a high level, the acquisition method uses the previously tested configurations as centroids for a neighborhood search. In particular, the neighboring configurations around the centroids are used as potential candidates from which a better configuration might emerge. This potential candidate set is then filtered using constraints on the predicted execution time, and a single configuration is selected randomly using a weighted probability. The job is then executed with this selected configuration, and its execution time is used to update the performance model.

5.2.1 Centroid selection. The algorithm selects previously executed configurations as *centroids* – all valid configurations with execution time within an ϵ factor of the current lowest execution time – whose neighboring configurations can provide better configurations.

The algorithm attempts to select at least l centroids. Given that fewer than l configurations may satisfy the ϵ-factor condition (especially early in the online optimization phase), the algorithm uses a scoring function to extend the selection to previously executed configurations that meet the execution time constraint and represent good choices in terms of their execution cost. Let ET_{min} be the lowest execution time, and EC_{max} be the highest cost across all executed configurations. A configuration C is scored as follows (higher is better):

$$score(C) = \begin{cases} \frac{(1+\epsilon)ET_{min}}{ET(C)} & EC(C) \leq EC_{max} \\ \frac{(1+\epsilon)ET_{min}}{ET(C)} + \frac{EC_{max}}{EC(C)} & EC(C) > EC_{max} \end{cases}$$

Intuitively, this mechanism assigns higher scores to configurations that satisfy cost constraints and lower scores to configurations that do not, proportionally to the amount by which they exceed the constraints. This ensures that, within the configurations that meet the time constraint, preference is given to searching the neighborhood of valid configurations (as far as their cost is concerned) with lower execution times, followed by configurations that violate the cost constraint with the lowest margin.

5.2.2 Candidate set selection. Based on the set of centroids O, the algorithm forms a *candidate set* of promising but untested configurations. This set is the union of configurations drawn from the neighborhood of each centroid and through instance type variations of the currently best configuration. Figure 4 shows an example of this process.

Neighborhood selection. For each centroid, the algorithm selects every configuration within distance d from the centroid. We define a distance metric based on the framework-wise difference of CPU and memory capacity. We say that the capacity $cap_F(C)$ of a configuration C w.r.t. framework F is the vector ⟨total CPU count, total memory⟩, where

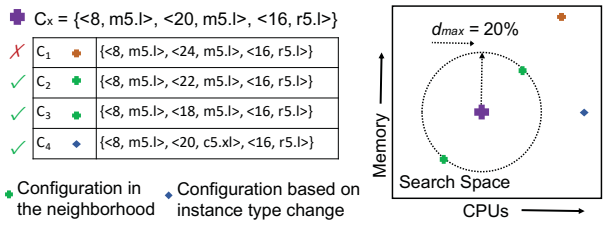

Figure 4: Example candidate set based on 20% maximum distance from a centroid (purple cross) for the middle framework (⟨20, m5.l⟩). The valid configurations in the neighborhood are those with a variation of ±10% memory and ±10% number of CPUs (combined ±20%). A selected configuration based on instance type variation doubles the number of CPUs and maintains the same memory.

total refers to the sum of all N_F instances of F. The distance between two configurations X, Y for framework F is $dist_F(X,Y) = ||cap_F(X) - cap_F(Y)||_1$. Therefore, so far the candidate set is $\bigcup_{o \in O}\{C \mid dist_F(C,o) \leq d, \forall F\}$.

The threshold d is set as $max(d_s, d_{max})$, where d_s denotes the distance of a single step away from centroid. This only applies when the hyperparameter d_{max}, which is specified in relative terms, overly restricts the candidate set. Note that because of our definition of distance, in theory, there could be configurations that differ from a centroid but their distance from it is zero. These configurations are also included.

Instance type variations. We observe that the above candidate set would rarely include an instance family different from those of the centroids because changing the instance family (for the same number of instances) can double or halve CPU and memory resources (we expect d_{max} to be smaller than 100%, see §5.3). To allow for instance family variations, we include in the candidate set configurations more distant than d_{max} from a centroid. Namely, the algorithm enumerates all possible instance type variations for the centroid configuration and includes those that do not reduce the total available memory. The intuition behind this is that changes in CPU resources can affect the job execution time, but drastic reductions to memory capacity can cause job failures. As reference centroid, the algorithm only considers the current best configuration, and all framework-wise instance type variations are considered. Since only the best configuration is used for this step, the possible decrease in the execution time is not severe.

5.2.3 Configuration selection. Using the candidate set Q, the algorithm selects a configuration in Q for use in the next production run. First, the algorithm filters the candidate set to discard any configuration whose predicted execution time is too high compared to the currently best configuration and the best-predicted execution time of the configurations in Q. Namely, $C \in Q$ is discarded if $t(C) > \theta \min_{X \in Q}(ET(X))$ or if $t(C) > \phi ET_{min}$. Here, θ and ϕ are hyperparameters whose tuning we describe in §5.3.

Next, the algorithm could pick from the remaining candidates the best configuration as predicted by the model. However, given the small number of executed configurations in the beginning, the model predictions may not be very accurate. Instead, the algorithm rates each configuration C according to its predicted execution time $t(C)$: $rate(C) = \max_{X \in Q}(t(X))/t(C)$; then the algorithm randomly picks the *next configuration* with a probability p following its normalized rating: $p(C) = rate(C)/\sum_{X \in Q} rate(X)$. This allows for some diversity while also biasing the choice to configurations with better predicted execution time.

5.3 Tuning the acquisition method

ϵ controls the closeness of the centroids to the best valid configuration in terms of the execution time. For instance, a value of 0.2 means that only points that have 20% higher execution time than the best-known execution time are chosen as centroids. l determines the desired number of centroids. d_{max} controls neighborhood size around the centroids. θ and ϕ limit the configurations that can be selected based on their predicted execution time w.r.t the best-predicted execution time in the candidate set and the best-known execution time. θ should be set slightly higher than ϵ to take into account the model prediction error. ϕ controls how conservatively the algorithm behaves; e.g., a value of 2 means that the algorithm will not select a configuration whose predicted execution time is twice the best-known execution time.

Higher values for all these hyper-parameters will allow for more exploration but might also lead to more execution time constraint violations. For our evaluation, we use the following values: $\epsilon = 0.2$, $\theta = 1.5$, $\phi = 2$, $l = 5$ and $d_{max} = 20\%$. These values worked well for our scenarios since, with these settings, the algorithm maintains a conservative approach in terms of the execution time constraint, while still being able to explore. A complete hyper-parameter exploration is left for future work.

6 LEARNING FROM OTHER JOBS

To try to reduce the number of benchmarking runs even further, Vanir uses the similarity between jobs and transfer learning to quickly bootstrap a performance model for the online optimizer, using knowledge from other jobs.

6.1 Telling jobs are similar

Before bootstrapping the performance model for a new job from a previous one, we need to first determine whether jobs are similar. Inspired by collaborative filtering algorithms [38], Vanir uses cosine similarity as a similarity metric applied to job *signatures*. We explored several types of signatures and

we settled on a simple one: a job signature is the pair of CPU %idle and memory utilization histograms. The histogram is created from a list of observations of the respective utilization metric. Observations from each instance of a framework are taken continuously and appended to create that list. In particular, these observations are collected every 5 seconds during a job run, and the histograms use bins at 10% granularity. Cosine similarity uses the concatenation of these two histograms in vectorized form.

We say that the pair of a new job and a prior job is similar when their similarity score is above a threshold (0.85 in our case). Vanir uses the model of the prior job with the highest similarity to bootstrap a performance model.

Unlike in [18, 19, 31, 43], we do not use collaborative filtering to get recommendations regarding the configurations based on prior jobs. This is because collaborative filtering requires certain sparsity conditions [16, 18, 19, 31], which might be impractical to achieve. Furthermore, these prior works rely on extensive offline profiling, in which a set of benchmark applications are profiled on all configurations. This is also impractical in our case, wherein the total number of possible configurations (Table 2) is roughly 900k. However, collaborative filtering or machine learning methods for workload characterization [18, 19, 29, 31, 41, 43] can be useful once there is a substantial history of a large number of jobs and job runs.

While cosine similarity between jobs allows bootstrapping models for unknown jobs, it is not a perfect method and may lead to false-positive matches. We add a simple test to avoid the effects of false positives: we measure the performance of the new job on the best and worst configuration of the most similar job, and, if the new job performs better on the best configuration from the known job compared to the worst configuration, then we bootstrap the model; otherwise, we fall back to the benchmarking phase. Thus, a successful application of similarity limits the number of benchmarking runs to a total of 3 (1 for similarity calculation and 2 to avoid false positives).

6.2 Bootstrapping models

Vanir uses transfer learning to adapt an MF model trained on a previously seen job and provide better results with fewer training samples for a new job. However, several challenges arise when putting this idea into practice. First, to the best of our knowledge, there is no existing transfer learning method for Mondrian Forests. Additionally, we want to minimize the number of samples needed from the new job to perform transfer learning. Therefore, we propose a simple technique that uses the difference in execution time between the two jobs (new job and a previously seen job) for the same configuration to offset the rest of the predictions of the model.

Parameters	Possible Values
Instance Type (in AWS EC2)	m5.large, m5.xlarge, c5.xlarge, c5.2xlarge, r5.large, r5.xlarge
No. of Instances (per framework)	[2, 32] - step size of 2

Table 2: Possible values of configuration parameters.

Assume that job J has the highest similarity with a new job H. In this case, we want to transfer J's performance model M_J to create a performance model M_H for H. To this end, we run H with the best configuration from model M_J and obtain the execution time ET_H. The difference $ET_H - ET_J$ between this execution time and the execution time obtained for the same configuration on J is then used as an offset to adjust the execution time predictions ($t_H(C)$) for M_H as: $t_H(C) = t_J(C) + ET_H - ET_J$ for all training configurations C used to build M_J.

7 EVALUATION

We evaluate Vanir against Baseline 1 and Baseline 2 (§2.4), which we implement atop the Spearmint [10] library for Bayesian optimization. All experiments run in AWS.

Table 2 lists the configuration search space, which yields 96 configurations for a single framework while the total number of possible configurations is 884,736. As such, it is impractical to search the entire space to determine the best configuration of each job as a ground truth.

We limit the optimization budget for each job to 18 runs in total. Baseline 1 uses 3 initial samples for each framework, while Baseline 2 uses a total of 6 initial samples (generated through Spearmint's default Sobol sampling).

7.1 Applications

We run a total of 9 realistic benchmarks (Table 1), 8 of which run on the analytics cluster shown in Figure 1. These benchmarks are adapted from applications included in the HiBench suite [6] and a few additional GraphX [4] jobs (sp, cc, lpa). To evaluate Vanir with *different pipelines*, we further run the price predictor (pp) job on a second cluster comprising a DAG of batch jobs. Our benchmarks, configs, and datasets are public at [8].

As for the dimension of input datasets, following the naming in HiBench, we use "gigantic" for pr, nw and lr; "huge" for gbt and rf; and "small" for sp, lpa, and cc. These sizes are chosen so that all jobs run in a reasonable amount of time (to reduce our costs) on at least a subset of the possible configurations. The datasets are generated synthetically using HiBench for each run.

We use the default values for all system parameters except for parallelism level and memory configurations for Spark.

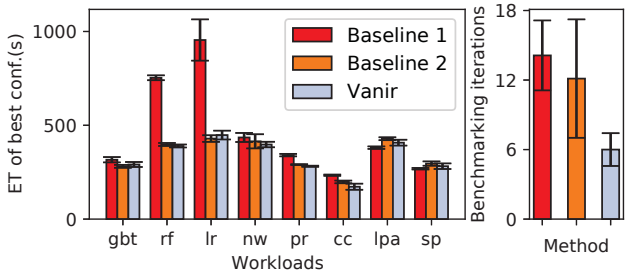

Figure 5: (a) Best execution time (left). (b) Number of benchmarking iterations (right).

Figure 6: Cost of optimization. Baseline 1 and 2 are generally 1.5-6 times more expensive than Vanir.

The parallelism level is set to 2× the number of cores available in the cluster allocated for Spark. In turn, the amount of executor memory and driver memory is set to the amount of available memory in the instances used for Spark.

7.2 Results

We compare Vanir against both baseline methods across: quality and cost of optimization; the speed of search; and the contribution of different components of Vanir. We also discuss the best configurations selected by Vanir and both baselines. We discuss the effects of the bounded search limitations of the baseline methods. Lastly, we generalize our results by looking at a different pipeline and variations in input dataset size.

7.2.1 Quality of optimization. We analyze the best configuration found by an optimization algorithm within a given search budget. In our case, this represents the best execution time found while satisfying the constraints.

Figure 5a shows the execution time of the best configuration found by each optimization method. We repeat the job with the best configuration for each method 5 times, and the error bars indicate the standard deviation. For all jobs, Vanir finds configurations that perform comparably to those by Baseline 2. Baseline 1, in turn, performs comparably to Vanir and Baseline 2 for 6 of the 8 jobs. The two jobs where Baseline 1 is outperformed are rf and lr. For these jobs, the best execution time found by Baseline 1 is 2-2.5× higher than Baseline 2 and Vanir. This is due to the limited view of the configuration space that Baseline 1 explores, since it optimizes one framework at a time, and decides on the best configuration for a given framework before advancing to optimizing the next one.

Figure 5b shows the number of benchmarking iterations required by each algorithm. For Vanir, that is simply the number of iterations of the offline optimizer. For Baseline 1 and 2, this is the number of runs to reach the best found configuration within the optimization budget. In practice, there is no way to know if an algorithm has yet reached the best configuration it will find. We, therefore, assume an oracle that predicts if the baselines have reached their best configuration. Therefore, this result shows the best-case scenario for the baselines. The results show that Vanir uses an average of 6 benchmarking iterations. In turn, Baseline 1 and 2 use 14 and 12 benchmarking iterations, respectively – thus, on average, requiring to wait at least 2 times as long before starting production runs. Using only 1/2 benchmarking iterations, Vanir finds configurations comparable to Baseline 2, thanks to online optimization in production runs.

7.2.2 Cost of optimization. It is important to consider the monetary cost of the (benchmarking) optimization phase since this consists of running an offline optimization algorithm, which costs money (and time) without doing any useful work. Ideally, this should be kept to a minimum.

Figure 6 shows the cost of optimization, normalized by the cost of the benchmarking phase of Vanir. The bars for Baseline 1 and 2 represent the total cost of the optimization over the budget of 18 iterations compared to the benchmarking phase of Vanir. Vanir (Full) includes both benchmarking and production phases. The cost of the production phase is only the extra cost for each iteration of the production phase compared to what would cost if one were to simply run Vanir's best found configuration.[2] The results show that Vanir incurs in substantially lower optimization costs, due to its virtuous combination of techniques, namely a quick estimate of a good configuration in the benchmarking phase followed by gradually improving the configuration at production time.

Using only the benchmarking phase of Vanir, the cost is up to 60% lower than running both the benchmarking and production optimization phases. However, that reduction in optimization cost comes at the expense of decreased quality of optimization, as shown later in this section. On the other hand, Baseline 1 and 2 are roughly 1.3-4.6× and 1.6-5.9× more expensive than Vanir (Full), respectively, for 7 out of 8 jobs. The exception is gbt, where Baseline 1 costs 5.9×, and Baseline 2 costs 24× more than Vanir. This is primarily

[2] As if an oracle revealed the best configuration that Vanir will find.

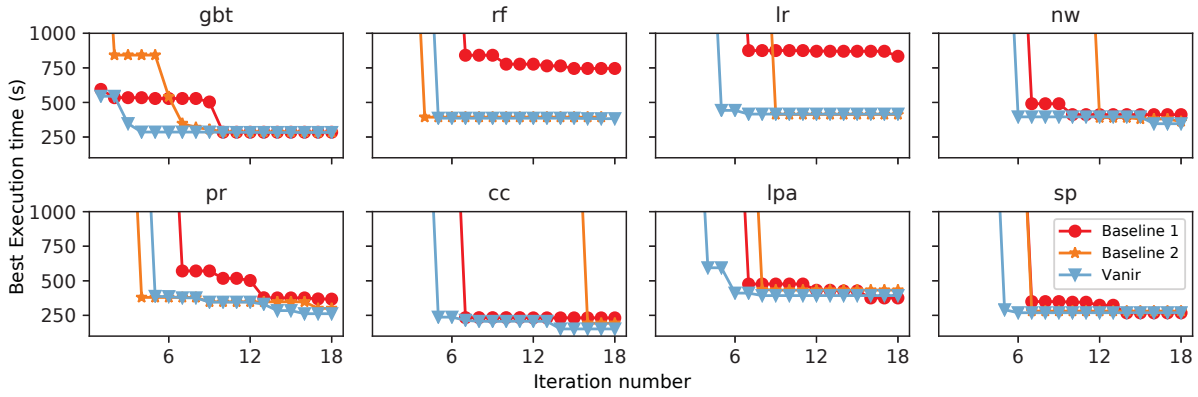

Figure 7: Progression of optimization for target jobs.

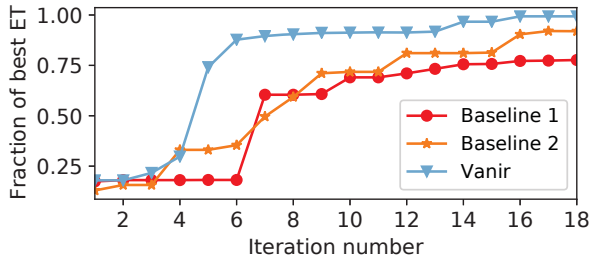

Figure 8: Speed of optimization. On average, Vanir takes only half the number of iterations compared to Baseline 2 to reach within 15% of the best execution time found.

because gbt does not require a lot of resources; in fact, bigger cluster sizes worsen its performance. Vanir's systematic approach with its benchmarking phase quickly determines that characteristic. Therefore, it avoids selecting larger configurations while the baselines often explore those regions of search space, thus incurring high optimization costs.

7.2.3 Speed of search. Another way to compare optimization algorithms is to evaluate which method is the fastest to reach a good configuration for a given search budget.

Figure 8 shows that on average, across the 8 jobs, how quickly each algorithm can get close to the best execution time found by any optimization method. The results show that Vanir is the fastest. On average, Baseline 2 takes 2× longer to get to a configuration within 15% of the best one, compared to Vanir. Because Vanir uses a metrics-based method as an offline optimizer, it keeps the benchmarking phase short by systematically changing the resources assigned to different frameworks instead of blindly generating initial samples, as it is the case for completely black-box methods. Vanir quickly finds a good configuration to start the production runs of the recurring job. Subsequent runs of the job can benefit from the production optimization phase.

Figure 7 shows the progression of the best-found execution time for each job separately. Vanir is fastest for 6 of the 8 jobs; except for rf and pr, whereby Baseline 2 by chance finds a valid configuration during initial sampling.

7.2.4 Constraint violations. Since Vanir performs optimization during production runs, it is worth evaluating the number of violations experienced (for time or cost constraints). Vanir assumes that cost constraint violations are tolerable during the production optimization phase. However, execution time constraint violations are more severe in production environments, and therefore an online optimizer should limit them. Any unsuccessful run of a job due to failure (e.g., memory exhaustion) is a violation.

Figure 9 shows the progression of execution time, and execution cost as Vanir optimizes the configuration for each job. The green circle indicates the best valid configuration found. The blue vertical line is the iteration number at which the benchmarking phase ends, and the production runs start. The black horizontal line is the cost constraint (associated with the right y-axis). Optimization runs that fail to execute or take longer than the execution time constraint are shown as iterations with zero execution time.

In our experiments, the online phase of Vanir leads to only one failure due to a lack of resources (for the iteration 11 of rf). However, if such an error occurs, the online optimizer quickly learns to avoid that region of the search space. It is clear that during the production phase, Vanir tests configurations that lead to cost constraint violations. This design is by choice, since making Vanir too conservative would hamper the quality of configurations it finds.

7.2.5 Contribution of different optimizers. To understand the contribution of Vanir's components to the overall optimization process, we break down the best execution time achieved by using different components. Figure 10 shows that the offline optimizer reaches performance close to the best configuration found by Vanir (Full), in 3 of the 8 jobs (sp, rf, and lr). In these cases, the online optimizer improves execution time by 0%, 1%, and 6%, respectively. In contrast,

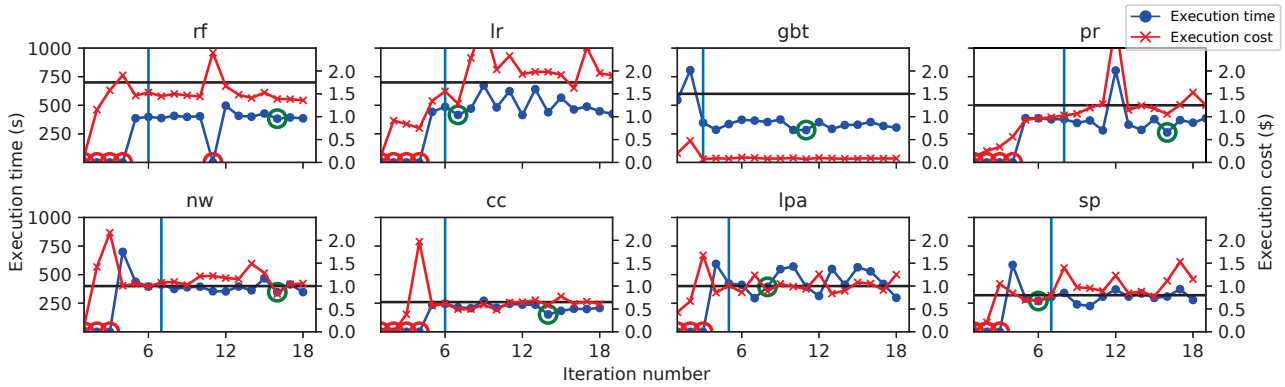

Figure 9: Progression of Vanir optimization across eight jobs. Only one ET constraint violation occurred in online phase.

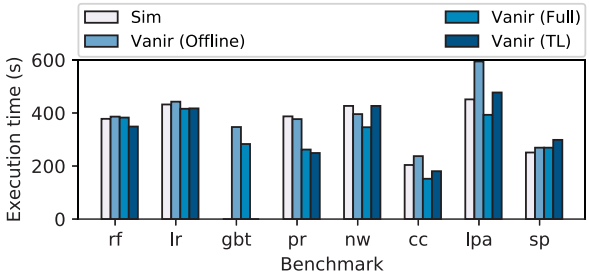

Figure 10: Contribution of different components of Vanir. Compared to offline-only optimization, the online phase lowers execution time by up to 36%.

the online phase improves execution time by 12-36% for the other 5 jobs.

7.2.6 Contribution of similarity and transfer learning. We now use a model learned on a job to optimize the configuration of another job. For a job J, using the best-known configuration from the most similar job leads to an execution time that is close to J's best-known execution time. Figure 10 shows that by using similarity (Sim), Vanir finds a configuration with an execution time that is at worst only 55% higher than the best-known configuration. However, these configurations do not always satisfy the execution cost constraints, a task handled by the production optimization phase that follows model bootstrapping.

Using transfer learning, we bootstrap a model for each job. Then we run the production optimization phase on the bootstrapped model to further improve performance. This is shown as Vanir (TL) in Figure 10. The improvement in execution time after running a production optimization phase on the bootstrapped model is 3.4%, 7.7%, 11%, and 35% for pr, rf, cc, and lr, respectively. There is no improvement in the case of nw. For sp and lpa, the execution time is worse than the one shown for *Sim*; that is because the configuration found using similarity did not satisfy the cost constraint, and so the configuration found after the production run phase in Vanir (TL) that satisfies the cost constraint has higher execution time. Interestingly, Vanir (TL) provides a trade-off between lowering the number of benchmarking runs and the quality of optimization. In 4 of the 7 cases, given the limited budget of 18 iterations, Vanir (TL) finds a configuration that is up to 20% slower than the one found by Vanir (Full). Hence, shortening the benchmarking phase is not always the right choice when the optimization quality is more important.

7.2.7 Unbounded search. Bayesian optimization, as used in Cherrypick [13] and Arrow [28], requires users to define the bounds of the optimization search space; specifically, the minimum and the maximum number of instances as well as possible instance types. While defining instance types is straightforward, defining the bounds on the number of instances is not easy without prior knowledge. While Bayesian optimization with Gaussian processes is limited to bounded search spaces, Vanir is not. This can be a useful property of our design, since defining a bound on the search space can be difficult. In particular, when defining this bound, there is a trade-off between the size of the search space and the cost of the optimization: a larger search space would likely require a larger budget and vice versa. Given that a user would likely not know about a job's performance profile across different configurations, it is easy to make mistakes when defining bounds on the search space.

Until now, we have been comparing both baseline methods with Vanir using a bounded search space, for a fair comparison. However, now we want to gauge the potential gains from an unbounded search; we keep the same bounds as in Table 2 for Baseline 2, while we set no upper limit for the number of instances of Vanir. Since the type and size of the instances would, in practice, be limited to a small number, we still keep the bounds on them. In this experiment, we use sp with the same dataset size used in previous experiments but with an execution cost constraint of $1.5 instead of $0.8 as used before.

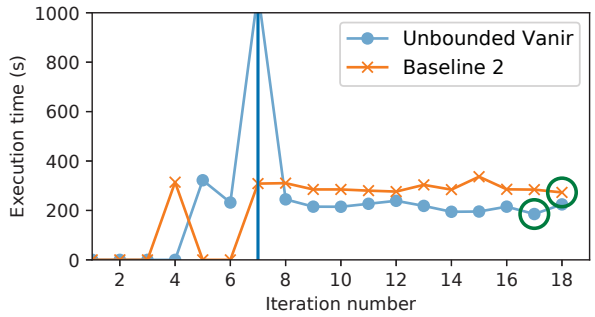

Figure 11: Comparing Vanir's performance on unbounded search space vs. Baseline 2 using sp**. Vanir automatically determines the right upper resource limits and avoids incorrect user-defined search space bounds.**

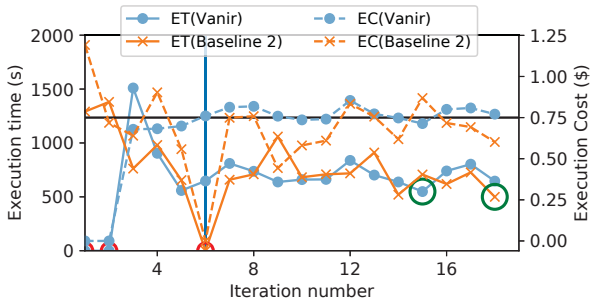

Figure 12: Comparing Vanir's performance on pipeline of batch jobs (pp**) vs. Baseline 2. Performance is comparable; Vanir spends just 6 runs in benchmarking phase.**

Figure 11 shows the performance of the configurations that both methods find. The results show that Vanir finds a valid configuration with an execution time of 185s, while Baseline 2 only finds a configuration with an execution time of 272s. This is because Vanir explores beyond the bounds that Baseline 2 is limited to. In particular, it tests configurations with 64 instances allocated to Spark and then also with 128 instances, and automatically finds that while 64 instances still provide a speedup, ~128 instances worsen the performance. Thus, unbounded search allows Vanir to find a configuration that is 32% faster than Baseline 2, for the same constraints and the same optimization budget.

7.2.8 In-depth look at the best configurations. We now want to reason about why one optimization algorithm performs better than another by taking a look at the best configurations found by each optimization algorithm. Answering this question can also help us understand the inefficiencies in each optimization algorithm. We will only discuss select few cases here.

Generally, HDFS and Cassandra are allocated fewer resources than Spark due to cost constraints. We also observe different best configurations across different workloads. HDFS and Cassandra are generally allocated a similar amount of resources except for the rf workload, where HDFS is given 2× resources compared to Cassandra (in the best configuration).

We find that Baseline 1 generally ends up only assigning more resources to Spark. In contrast, the best configurations found by both Vanir and Baseline 2 assign more resources to other frameworks as well. Thus, Baseline 1 misses the opportunities that the other two methods explore (since they perform joint optimization). This is particularly important for rf and lr, and that is why Baseline 1 performs poorly compared to the other two methods for those jobs.

In addition, we find that different instance types matter. Specifically, Vanir's best configurations for pr and gbt jobs use c5.2xlarge and c5.xlarge instances, respectively. These changes improve the execution time by 18% and 30% (compared to using m5 instances) for gbt and pr, respectively. Interestingly, none of the best configurations use the maximum allowed resources for every framework because: (1) the execution cost constraint means that even if higher resources decrease the execution time, they might incur a high execution cost, and (2) in some cases, assigning more resources can degrade execution time; e.g., in case of gbt.

7.2.9 Generalizing to a pipeline of batch jobs. To validate that Vanir's approach applies to other types of data analytics pipelines, we use a pipeline of batch processing jobs. This kind of pipelines is common in industry use case [2, 5, 9], enabled by tools such as Apache Airflow [1] and Luigi [7], which have been designed to facilitate data pipelines in the form of a DAG of batch jobs. In our case, the pipeline consists of a linear DAG of three Spark jobs [8] wherein the results of one job serve as input to the next.

Figure 12 shows the comparison between Vanir and Baseline 2 for this data pipeline. Both methods eventually find the best configurations (highlighted with green circles) that perform comparably. However, Vanir does not have significant performance variations during the production runs (beyond the vertical blue line) and thus it is more suitable for optimization using production runs. In contrast, Baseline 2 only reaches its best configuration at the end of its offline optimization budget (18 iterations). For this scenario, Vanir determines that the Update stage requires more resources than the other two stages.

7.2.10 Handling input data size changes. Since Vanir performs part of the optimization using production runs, one could expect that variations in the input data size for different job invocations affect performance. Vanir handles these changes (whenever input data size changes significantly, say at least a 10% change) by adapting the job's performance model to different input data sizes, treating the latter as an extra model feature. The initial configuration used is a linearly

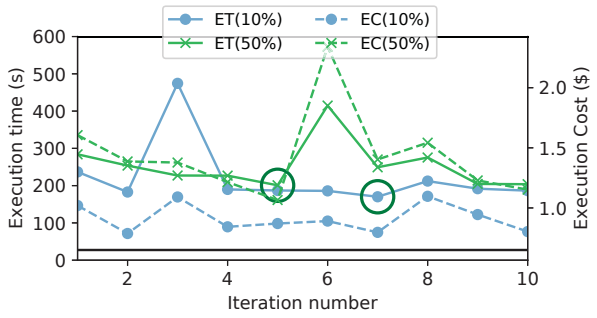

Figure 13: Handling changes in input data size for cc**. Vanir improves the execution time by 30% within 10 optimization runs for input data size increase of 10% and 50%.**

and proportionally-scaled version of the best configuration for the original input size.

Figure 13 shows the progression of optimization of cc over 10 iterations when the data size is 10% and 50% higher than the data size on which the original cc model was created. Vanir finds configurations (highlighted with green circles) that provide 170s and 200s execution time for input data with 10% and 50% higher data size, respectively. This represents an improvement of 30% over the initial (proportionally larger configuration).

8 RELATED WORK

Cloud configuration optimization. Ernest [42] adopts an analytical model to predict the execution time of Spark jobs, mainly targeting ML applications. The model is limited to Spark and is not applicable for the performance prediction of multi-framework analytics clusters. Elastisizer [26] uses a mix of black-box and white-box models to optimize cloud configurations as well as job-level configurations for map-reduce jobs. However, several of the models used are limited to the map-reduce style of jobs.

Cherrypick [13], Arrow [28], Micky [27] and Lynceus [17] treat the data analytics framework as a black box, and perform cloud configuration optimization using offline methods. These initial proposals targeted a single framework and therefore only had to consider a configuration search space on the order of tens of configurations. We build on that line of research but consider a much larger search space, where a completely offline method becomes impractical. Instead, Vanir splits the optimization into offline and online phases.

Scout [29], Selecta [31] and PARIS [43] use historical data to quickly choose cloud configurations for new jobs. Selecta incorporates different storage options into the cloud configuration search space. Lessons learned from Selecta can be incorporated into Vanir's offline optimizer. The increased size of the search space as a result of including storage options would make Vanir even more attractive. However, these were designed for and evaluated in single system deployments, and the amount of historical data required for handling clusters with multiple systems might be impractical. Additionally, given their different target deployment, Scout and PARIS did not need to improve the configuration suggested by historical data at deployment time, which Vanir does through online optimization.

System configuration optimization. Research works in configuration optimization tackle an orthogonal problem of optimizing system configurations such as parameter tuning for Hadoop [14, 25], Spark [21, 23], Storm [15, 40] or databases [22, 34] or general-purpose frameworks like BestConfig [44]. Tools such as LearnConf [33] can be used to determine the most important system parameters that impact the performance and guide the tuning.

Resource allocation in data centers. Paragon [18] is a data center heterogeneity and interference-aware scheduler. It uses collaborative filtering to classify an unknown incoming job to assign resources to it. Quasar [19] is a follow-up work on Paragon, which also uses collaborative filtering for unknown workload classification. Despite this being a different problem scenario, we note that a downside of collaborative filtering requires an offline training set that is impractical to obtain when the configuration space is large. DejaVu [41] is another work that tackles the problem of allocating resources to workloads in a datacenter setting. Their method of creating workload signatures might be a complementary alternative to the simple similarity metric used by Vanir when there is a large set of jobs and performance data. Morpheus [30] is designed to provide high cluster utilization and predictable performance for enterprise clusters. It extracts SLOs implicitly and performs job placement and dynamic re-provisioning to achieve the SLO. In contrast to Vanir, Morpheus is designed for a data center environment, where there are no cost constraints or considerations of heterogeneous types of machines. Perforator [35] introduces a methodology to perform resource optimization for Hive. Several analytical models are designed and used in that work, but they are not applicable in our setting with multi-frameworks.

9 CONCLUSION

Vanir performs automatic cloud configuration optimization by splitting the optimization task into benchmarking and production phases. This allows Vanir to handle large configuration search spaces without requiring long offline benchmarking or optimization. Vanir finds configurations comparable to benchmarking based offline methods despite a 3× shorter benchmarking phase. Vanir has 1.3-24× lower optimization search cost, and it is 2× faster to reach within 15% of the execution time of the best configuration found.

ACKNOWLEDGMENTS

Muhammad Bilal was supported by a fellowship from the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC) program funded by the European Commission (EACEA) (FPA 2012-0030). This research was supported by Fundação para a Ciência e a Tecnologia (FCT), under projects UIDB/50021/2020, CMUP-ERI/TIC/0046/2014, and POCI-01-0247-FEDER-045915.

REFERENCES

[1] Apache Airflow. https://airflow.apache.org. [Online; accessed 25/05/2020].
[2] Build a Concurrent Data Orchestration Pipeline Using Amazon EMR and Apache Livy. https://aws.amazon.com/blogs/big-data/build-a-concurrent-data-orchestration-pipeline-using-amazon-emr-and-apache-livy/. [Online; accessed 25/05/2020].
[3] Flink Use cases. https://flink.apache.org/usecases.html. [Online; accessed 01/03/2020].
[4] GraphX lib github. https://github.com/apache/spark/tree/master/graphx/src/main/scala/org/apache/spark/graphx/lib.
[5] How Verizon Media Group migrated from on-premises Apache Hadoop and Spark to Amazon EMR. https://aws.amazon.com/blogs/big-data/how-verizon-media-group-migrated-from-on-premises-apache-hadoop-and-spark-to-amazon-emr/. [Online; accessed 25/05/2020].
[6] Intel's HiBench benchmark. https://github.com/intel-hadoop/HiBench.
[7] Luigi. https://github.com/spotify/luigi. [Online; accessed 25/05/2020].
[8] Modified version of Intel's HiBench benchmark. https://github.com/MBtech/HiBench.
[9] Quoble Pipeline. https://www.qubole.com/developers/spark-getting-started-guide/workflow/. [Online; accessed 25/05/2020].
[10] Spearmint Github repo. https://github.com/HIPS/Spearmint.
[11] Walmart Labs: Lambda Architecture. https://medium.com/walmartlabs/how-we-built-a-data-pipeline-with-lambda-architecture-using-spark-spark-streaming-9d3b4b4555d3. [Online; accessed 01/03/2020].
[12] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Re-optimizing data-parallel computing. In *NSDI*, pages 21–21. USENIX Association, 2012.
[13] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *NSDI*, 2017.
[14] S. Babu. Towards Automatic Optimization of MapReduce Programs. In *SoCC*, pages 137–142. ACM, 2010.
[15] M. Bilal and M. Canini. Towards automatic parameter tuning of stream processing systems. In *SoCC*, pages 189–200. ACM, 2017.
[16] F. Cacheda, V. Carneiro, D. Fernández, and V. Formoso. Comparison of collaborative filtering algorithms: Limitations of current techniques and proposals for scalable, high-performance recommender systems. *ACM Transactions on the Web (TWEB)*, 5(1):2, 2011.
[17] M. Casimiro, D. Didona, P. Romano, L. Rodrigues, W. Zwanepoel, and D. Garlan. Lynceus: Cost-efficient tuning and provisioning of data analytic jobs. *arXiv preprint arXiv:1905.02119*, 2019.
[18] C. Delimitrou and C. Kozyrakis. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. In *ASPLOS*, pages 77–88. ACM, 2013.
[19] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and QoS-aware Cluster Management. In *ASPLOS*, pages 127–144. ACM, 2014.
[20] M. Denil, D. Matheson, and N. Freitas. Consistency of online random forests. In *ICML*, pages 1256–1264, 2013.
[21] H. Du, P. Han, W. Chen, Y. Wang, and C. Zhang. Otterman: A novel approach of spark auto-tuning by a hybrid strategy. In *ICSAI*, pages 478–483, 2018.
[22] S. Duan, V. Thummala, and S. Babu. Tuning Database Configuration Parameters with iTuned. In *PVLDB*, pages 1246–1257. VLDB Endowment, 2009.
[23] A. Fekry, L. Carata, T. Pasquier, A. Rice, and A. Hopper. Tuneful: An online significance-aware configuration tuner for big data analytics. *arXiv preprint arXiv:2001.08002*, 2020.
[24] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *EuroSys*, pages 99–112. ACM, 2012.
[25] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. In *PVLDB*, pages 1111–1122. VLDB Endowment, 2011.
[26] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *SoCC*, page 18. ACM, 2011.
[27] C. Hsu, V. Nair, T. Menzies, and V. Freeh. Micky: A cheaper alternative for selecting cloud instances. In *CLOUD*, volume 00, pages 409–416. IEEE, 2018.
[28] C.-J. Hsu, V. Nair, V. W. Freeh, and T. Menzies. Arrow: Low-Level Augmented Bayesian Optimization for Finding the Best Cloud VM. In *ICDCS*, pages 660–670. IEEE, 2018.
[29] C.-J. Hsu, V. Nair, T. Menzies, and V. W. Freeh. Scout: An experienced guide to find the best cloud configuration. *arXiv preprint arXiv:1803.01296*, 2018.
[30] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Krishnan, J. Kulkarni, et al. Morpheus: Towards automated slos for enterprise clusters. In *OSDI*, pages 117–134. USENIX Association, 2016.
[31] A. Klimovic, H. Litz, and C. Kozyrakis. Selecta: Heterogeneous cloud storage configuration for data analytics. In *USENIX ATC*, pages 759–773, 2018.
[32] B. Lakshminarayanan, D. M. Roy, and Y. W. Teh. Mondrian forests: Efficient online random forests. In *NIPS*, pages 3140–3148, 2014.
[33] C. Li, S. Wang, H. Hoffmann, and S. Lu. Statically inferring performance properties of software configurations. In *EuroSys*, pages 1–16, 2020.
[34] A. Mahgoub, P. Wood, S. Ganesh, S. Mitra, W. Gerlach, T. Harrison, F. Meyer, A. Grama, S. Bagchi, and S. Chaterji. Rafiki: a middleware for parameter tuning of NoSQL datastores for dynamic metagenomics workloads. In *Middleware*, pages 28–40, 2017.
[35] K. Rajan, D. Kakadia, C. Curino, and S. Krishnan. PerfOrator: Eloquent Performance Models for Resource Optimization. In *SoCC*, pages 415–427. ACM, 2016.
[36] D. M. Roy, Y. W. Teh, et al. The mondrian process. In *NIPS*, pages 1377–1384, 2008.
[37] A. Saffari, C. Leistner, J. Santner, M. Godec, and H. Bischof. On-line random forests. In *ICCV*, pages 1393–1400. IEEE, 2009.
[38] B. M. Sarwar, G. Karypis, J. A. Konstan, J. Riedl, et al. Item-based collaborative filtering recommendation algorithms. In *WWW*, volume 1, pages 285–295, 2001.
[39] L. Shao, Y. Zhu, S. Liu, A. Eswaran, K. Lieber, J. Mahajan, M. Thigpen, S. Darbha, S. Krishnan, S. Srinivasan, C. Curino, and K. Karanasos. Griffon: Reasoning about Job Anomalies with Unlabeled Data in Cloud-Based Platforms. In *SoCC*, pages 441—-452. ACM, 2019.
[40] M. Trotter, T. Wood, and J. Hwang. Forecasting a Storm: Divining Optimal Configurations using Genetic Algorithms and Supervised Learning. In *ICAC*, pages 136–146. IEEE, 2019.
[41] N. Vasić, D. Novaković, S. Miučin, D. Kostić, and R. Bianchini. Dejavu: accelerating resource allocation in virtualized environments. In *ASPLOS*, pages 423–436. ACM, 2012.

[42] S. Venkataraman, Z. Yang, M. J. Franklin, B. Recht, and I. Stoica. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *NSDI*, pages 363–378. USENIX Association, 2016.

[43] N. J. Yadwadkar, B. Hariharan, J. E. Gonzalez, B. Smith, and R. H. Katz. Selecting the Best VM Across Multiple Public Clouds: A Data-driven Performance Modeling Approach. In *SoCC*, pages 452–465. ACM, 2017.

[44] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In *SoCC*, pages 338–350, 2017.