PerfOrator: eloquent performance models for Resource Optimization

Kaushik Rajan, Dharmesh Kakadia, Carlo Curino, Subru Krishnan
Microsoft
[krajan, dkakadia, ccurino, subru]@microsoft.com

Abstract

Query Optimization focuses on finding the best query execution plan, given fixed hardware resources. In BigData settings, both pay-as-you-go clouds and on-prem shared clusters, a complementary challenge emerges: *Resource Optimization: find the best hardware resources, given an execution plan*. In this world, provisioning is almost instantaneous and time-varying resources can be acquired on a per-query basis. This allows us to optimize allocations for completion time, resource usage, dollar cost, etc. These optimizations have a huge impact on performance and cost, and pivot around a core challenge: faithful resource-to-performance models for arbitrary BigData queries. This task is challenging for users and tools alike due to lack of good statistics (high-velocity, unstructured data), frequent use of UDFs, impact on performance of different hardware types and a lack of understanding of parallel execution at such a scale.

We address this with *PerfOrator*, a novel approach to resource-to-performance modeling. *PerfOrator* employs non-linear regression on profile runs to model arbitrary UDFs, calibration queries to generalize across hardware platforms, and analytical framework models to account for parallelism. The resulting estimates are orders of magnitude more accurate than existing approaches (e.g, Hive's optimizer), and have been successfully employed in two resource optimization scenarios: 1) optimize provisioning of clusters in cloud settings—with decisions within 1% of optimal, 2) reserve skyline of resources for SLA jobs—with accuracies over $10\times$ better than human experts.

Categories and Subject Descriptors C.2.4 [*Computer-Communication Networks*]: Distributed Systems—Distributed databases

SoCC '16, October 05 - 07, 2016, Santa Clara, CA, USA.

ISBN 978-1-4503-4525-5/16/10. . . $15.00.
DOI: http://dx.doi.org/10.1145/2987550.2987566

1. Introduction

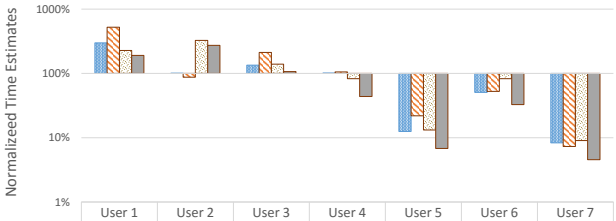
Query optimization has been central to much of database literature and system building, due to its substantial impact on query performance [14]. Slow provisioning cycles made hardware a "constant" in the formulation of this problem—which focuses on finding the best query plan, given *fixed* hardware resources.

In the context of BigData, two recent trends (clouds and large, on-prem clusters) are challenging this assumption. The advent of cloud offerings, such as Azure and AWS, with their pay-as-you-go model, drastically shortens the hardware provisioning cycle from months to minutes, as well as the typical horizon for ownership, from 3+ years to as little as 1 hour. This allows attentive users to purchase access to different (and best fitting) hardware resources [3] for each query of an analytics workload[1]. As an example, we show that running the BigBench workload [15] at 10TB scale on an optimal, fixed cluster of VMs is on average 45% more expensive than picking the best configuration for each query individually. For on-premise shared clusters, we observe a similar, yet even more dramatic trend. Resource managers such as Hadoop/YARN [31], Cosmos [6], Borg [34], and Mesos [19] effectively multiplex the access to massive clusters (thousands of machines) across queries within seconds. YARN [10] enables users to reserve resources ahead of job execution. Reservations take the form of *time-varying skylines*, that define how many containers[2] a job will require during every second of execution. They can range from few seconds on a single core to hours on thousands of machines. In this new world, properly selecting hardware can have a dramatic impact on query performance and cost.

Moreover, users today lack basic system support to explore how resources affect query performance. Our hunch is that users have a hard time picking among simple alternatives, and that deriving resource skylines is even harder. We put this to the test with a simplified A-vs-B experi-

[1] Note that in these settings persistent data (like an input table) is often stored in a separate but collocated storage layer enabling compute VMs to be spun up and shutdown quickly without having to migrate data.

[2] A container in YARN is a collection of physical resources like memory, CPU and disk.



	User 1	User 2	User 3	User 4	User 5	User 6	User 7
Q1 on CA	299%	101%	134%	101%	12%	51%	8%
Q1 on CB	525%	88%	212%	105%	22%	53%	7%
Q3 on CA	227%	326%	140%	83%	13%	83%	9%
Q3 on CB	191%	273%	107%	44%	7%	33%	5%

Figure 1. User estimates for execution time of BigBench query Q1 and Q3 at 10TB scale on two Azure clusters: CA (128× VMs of type D3) and CB (128× VMs of type D12).

ment. We asked seven BigData engineers and scientists to choose between two Azure-based cluster configurations for running query Q1 and Q3 from BigBench [15] (our running examples throughout the paper). Users were given a HiveQL query formulation, the execution plan, and the actual data-sizes after each stage of the execution (to simulate extra prepwork and potential prior knowledge), and were asked to rank the two clusters options for each query as well as estimate runtime (and thus cost) of each alternative. Figure 1 shows the results of their work. The relative errors range from 20× under- to 5× over- estimations of the runtime and cost. Even worse, only 1 user (User 2) would have correctly selected cluster CB for query Q1 and cluster CA for query Q3, all others made at least one mistake. If expert users confronted with a simplified version of the modeling task find it so challenging: *How bad would normal users do? How hard is it to derive complete skylines?* We answer these questions further in our evaluation in § 7, and by analyzing users' behaviors on production clusters. The results are consistent with our worst fears, users systematically mis-provision the majority of jobs, frequently by 10× or more. We conclude that, in BigData settings, assuming hardware as a constant or that users can easily answer provisioning questions is simply unacceptable, and the emerging *Resource Optimization* problem deserves attention.

Our Proposal In this paper we formalize the resource optimization problem (§ 2) and present a novel solution to it. In addressing this problem we make a basic observation: the range of possible optimizations is broad and scenario specific (e.g., minimize execution time, or dollar cost, maximize cluster utilization, ensure job completion by a deadline, etc.), but all these optimizations pivot around one common challenge: *model the resource-to-performance behavior of queries*. To address this we design *PerfOrator* an "*eloquent*" performance model that, given a hypothetical hardware configuration, produces a resource consumption skyline.

Resource-to-performance modeling of BigData queries is hard for two reasons. First, the performance of a multi-stage query is governed by the volume of data processed and transferred by each stage (processing more data requires more resources, takes longer). Estimating data-sizes for a BigData query that uses arbitrary UDFs/UDAs, and processes data that lacks statistics is hard (See § 2 for details). Second the time to process data depends on the performance characteristics of hardware resources and how the execution framework exploits parallelism to overlap the access to different resources. Scalable and accurate models are needed for estimation in such a massively parallel setting.

PerfOrator uses two estimators. 1) The *Data-size Estimator* employs non-linear regression over few profile runs on sample data (no data statistics needed) to model relational operators and UDFs alike. 2) The *Performance estimator* uses (a) analytical models to reason about parallelism optimization of BigData execution frameworks and (b) hardware calibration to generalize time estimates over different cluster configurations.

We showcase the flexibility of our models (§ 6), by embedding them in two scenario-specific optimizers: 1) provisioning of cloud-based clusters, and 2) reservation of resource skylines for SLA jobs in on-prem settings [10].

We then thoroughly validate its performance on popular benchmarks [15] and production workloads from Microsoft. The results (§ 7) are very encouraging, *PerfOrator*'s data size estimator systematically outperforms production systems like Hive by up to 7 orders of magnitude. It provides very accurate runtime estimates, within a factor 2, for all tested queries. At provisioning cluster on the cloud, *PerfOrator* is on average within 0.6% of optimal. By contrast, baseline approaches or optimal, fixed strategies incur extra costs of 45% or more. Moreover, *PerfOrator* performs 13x better than an expert at the most challenging Resource Optimization task: skyline prediction.

Limitations Our solution solves Resource Optimization as an orthogonal problem to Query Optimization. We do so to highlight the potential benefits of resource optimization in the BigData setting. Our solution is broad and can also be applied directly to resource optimize hand coded non-declarative BigData jobs. Designing a joint optimizer is part of our future research agenda.

2. Problem Settings

The problem can be formulated as follows:

Resource Optimization: *given a fixed computational task and input data, select a bounded, time-varying allocation of hardware resources that maximizes an objective function f, where f is a function of the task's performance and resource consumption.*

In tackling this problem we assume the following:

1. The computational task, also referred to as a query or job, is fully defined (e.g., a query with a chosen execution plan, and known input data.).

2. The hardware alternatives are well-defined and bounded (e.g., ability to assign up to n VM instances from k known types, in each of t time intervals.).

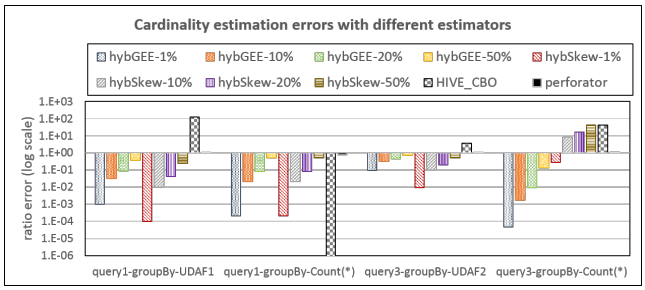


Figure 2. Comparison of cardinality estimates for UDA's and deep pipelines. The figure shows the ratio error (predicted/actual) for different estimators. HybGEE-x and HybSkew-x are sampling based distinct value estimators (x is the sampling rate). HIVE_CBO is the statistics based estimator used by Hive. PerfOrator is our system. Note : errors are plotted in log scale.

3. Our system is allowed to perform off-line calibration on each hardware type (e.g., ran a few purpose-built queries on each VM type.).

4. Our system has access to bounded amounts of resources of *one* type at optimization time (e.g., to profile the query on sample data.).

A solution to this problem is likely to consist of: (i) a faithful resource-to-performance model and (ii) an algorithm that repeatedly invokes this model to explore the $O(n * k * t)$ solution space, to optimize the function f.

Solution alternatives Next we contrast our solution with existing approaches in related settings. Our solution to (i) builds on two components, a *data size estimator* and a *performance estimator*, we begin with these.

Data size estimation The data size estimation problem is similar to the problem of cardinality estimation, one that is studied richly in the context of relational databases [14]. The problem of data size estimation for BigData queries is especially hard because of the common use of UDFs and UDAs besides more traditional relational operators. Our analysis of a five month long trace of production Hive sessions reveals that about 40% of the queries use UDFs. This is consistent with previous studies [5]. Second, even with SQL operators, deep pipelines, complex predicates and correlation among data produce compounded errors of up to multiple orders of magnitude (see [26] for a recent analysis). Third, most cardinality estimation techniques rely on having good statistics on the input data. Such statistics may not be available as data is of most value as it arrives (gathering statistics can take hours) and is often stored in formats that require preprocessing (decompression/deserialization) as part of query execution. Last, since performance is affected by physical IO (and not logical record count), we must focus on actual data-size vs cardinality, to account for storage formats and compression.

In Figure 2 we show the error in cardinalities predicted by the Hive cost-based optimizer [2] for the example queries. It also compares with distinct-value estimators from literature [7, 16] when applied directly to UDA's and deep queries. We observed orders of magnitude errors for most practical cases. We need to push sample sizes for the state-of-the-art approaches all the way to 50% (impractical) to obtain estimates within a factor 2. To address these challenges, we propose a new regression based data-size estimator, discussed in § 4. Figure 2 also shows our approach, with estimation errors of less than 10% at sample sizes of 1,2 and 3%.

Performance estimation Estimating performance in this context is particularly hard, because a BigData job runs on a massively parallel system. In particular for resource optimization estimation across different cluster types (without profiling the query on each) and modeling different frameworks is non-trivial.

Recent related work [11, 22, 25, 33, 35] has recognized the relevance of this problem and achieved solid results on slightly simplified variants: runtime estimation on the same hardware where queries are profiled, or for single VM execution of queries on different cloud VM types, instead of full-skyline estimation on different hardware for arbitrarily deep SQL-based DAGs. A complete discussion of related work appears in § 8. *PerfOrator* addresses these challenges using a gray box model that offloads query independent modeling to offline calibration (see § 5).

Searching over a solution space Our solution to (ii) above is related to the problem of finding an optimal query plan over a space of possible solutions. Our current search algorithms avoid exploring a large space by exploiting scenario specific strategies (more details in § 6). In particular they avoid having to explore over the dense time dimension (t) making the problem more tractable. In future when we integrate query and resource optimization we hope to leverage work on parametric and multi-objective query optimization [20, 28, 30] to efficiently search the combined space.

3. Overview

In this section we provide an overview of our solution. The path of a query (Figure 3) through the system is as follows: (step 1) the query is fed to an existing query-optimizer, (step 2) the resulting execution plan is analyzed to derive the resource-to-performance model, (step 3) the model is used to pick the best resource allocation and to enforce it by interacting with the underlying resource management layer.

Step 1 This step is language/application specific. Our current implementation leverages Hive's query optimizer, and its pre-execution hooks to integrate with Step 2. We are currently exploring integration with Cosmos [6] and Spark [37].

Step 2 This step forms the core of this paper. *PerfOrator* consists of two parts, a data-size estimator (§ 4), and a performance estimator (§ 5) as shown in Figure 3. The various stages are discussed in more detail below:

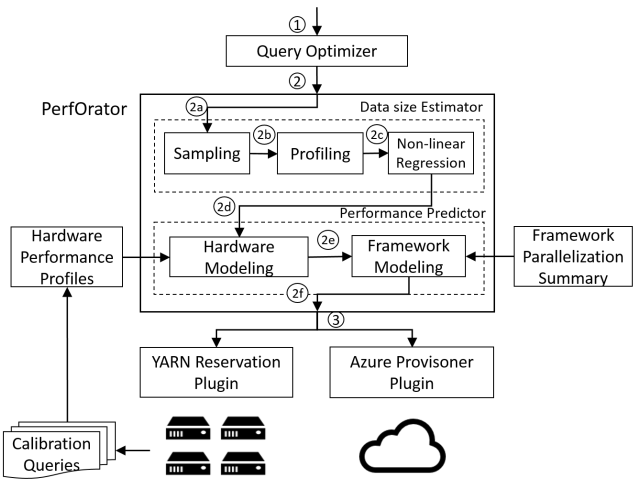


Figure 3. Resource optimization flow

Sampling (online, cacheable) The input tables of the query are identified using a semantics analyzer and samples of different sizes for such tables are created if they do not already exist. The cost of sampling is amortized by reusing samples across queries.

Profiling (online) The query is rewritten to run on the samples of the input tables and the data sizes at the input and output of each stage are collected. The CPU cycles spent processing each stage are also collected. Note that, in settings where online profiling is deemed too expensive, *PerfOrator* can leverage historical models or statistics based cardinality estimators. We showcase our integration with Hive's cardinality estimator and its impact on quality in § 7.

Non-linear regression (online) Non-linear regression is used to learn a function that relates input to output data sizes for each stage, as well the relationship between the input data size of a stage and the CPU cycles required. Once the functions are learned from the samples, the system propagates the sizes for the original query throughout all the stages, and thus estimates the full-scale data sizes and computation requirements for all stages of the query.

Hardware modeling (predominantly offline) Hardware performance profiles are used to translate data-sizes in bytes into time components in seconds. In particular they are used to estimate the time needed for reading, writing and shuffling data in each stage. They also translate estimated compute time on the profile run to corresponding estimates on alternative hardware configurations. The performance profiles are themselves gathered offline in a black box manner by employing calibration queries for different hardware resources (memory, disk, HDFS, Azure store, network, CPU) on each cluster type. The calibrations queries are run many times to estimate variance in hardware performance.

Framework modeling (online) *PerfOrator* employs white-box analytical models to reason about the effectiveness of parallelization optimizations used by different execution frameworks. The models take as input a parallelization summary for an execution framework (constructed offline from performance logs and through code inspection) and determine which time components are critical and contribute to execution time of the stage and which can be hidden due to parallelism. It estimates both the mean and variance in execution time of each stage. The model also estimates the degree of parallelism by mimicking the algorithm that the execution framework uses. We currently target MapReduce and Tez execution frameworks. Note that the models are abstract and easily extensible to other frameworks (§ 5.3).

The above process terminates once a complete model for the query is constructed, this model can then be cheaply probed ($< 1ms$) to explore the performance of a query on different type and amount of hardware resources.

Step 3 This step is specific to the optimization one wants to perform (§ 6). We implement two variants integrating *PerfOrator* with Azure's resource provisioner, and YARN's reservation subsystem [10]. Our cloud integration module allows users to specify a range of hardware configurations to consider, and asks the system to minimize either the total dollar cost or execution time of queries. The integration with YARN is designed to provide the skyline of a job's resource demand. This is used by YARN's reservation system API, that, in turn, ensures enough resources are booked for the job to complete within a user-specified SLA [10].

4. Data size estimation

Algorithm 1 shows the pseudo code for our data size estimator. The code has three parts corresponding to the sampling, profiling and regression steps. We begin with a description of the regression step which forms the core of our technique. We describe the sampling and profiling steps later in Sections 4.2 and 4.3 respectively.

4.1 Non-linear regression

Non linear regression is a statistical process for estimating the relationship between input variables and their dependent observed variables. Given a function form F with some unknown variables statistical techniques are used to find the values for the unknowns that fit the observations best. In our setting we are interested in discovering the relationship between the data sizes of successive stages. That is the relationship between the size of the input to a stage and its output, and also between the size of output of previous stages and the input to a stage. In this paper we explore functions of the form $output = b(input)^c$ where $input$ is a measure of stage input, $output$ is the observed variable representing the output data size, b and c are the unknown parameters. This function can fit super-linear, sub-linear and linear relationships. We show in § 4.1.1 that this captures the input-output relations that we expect relational operators and UDFs to exhibit. In addition this function has two other desirable features.

Algorithm 1 DataSizeEstimator (Q, sampleRates)

```
1:  inputTables = getAllInputTables(Q)
2:  fn = Function("b ∗ x^c")
3:  for s in sampleRates do
    // Generate samples
4:      for t in inputTables do
5:          createSampleTableIfNotExist(t,s)
6:      end for

    // Profile query on samples of data
7:      Q_s = rewriteQuery(Q,s)
8:      P_s = runQuery(Q_s)
9:      annotatedPlan = annotateQuery(Q, P_s)
10: end for

    // Estimate data sizes
11: for v in topDownTraversal(annotatedPlan) do
      // Estimate data transferred along each edge
12:     for u in v.inputVertices do
13:         params = regression(v.inSizes, u.outSizes, fn)
14:         Est_inSize + = eval(fn, u.Est_inSize, params)
15:     end for

        //Estimate output size from input
16:     params = regression(v.inSizes,v.outSizes,fn)
17:     Est_outSize = eval(fn, Est_inSize, params)

        //Estimate CPU cycles
18:     params = regression(v.inSizes,v.cpuCycles, fn)
19:     Est_cpuCycles v = eval(fn,Est_inSize, params)
20:     annotatedPlan = annotateQuery(annotatedPlan, Est)
21: end for
22: return annotatedPlan
```

Simplicity The overhead of the estimator depends on the number of samples needed. The more the unknowns the larger the number of samples needed and larger the overhead. This function has only two unknowns and, as we demonstrate through experiments, we make good predictions with just three observations.

Composabilty We are only interested in data sizes at stage boundaries. The desired relationship is just a composition of the functions for individual operators. Composability allows us to directly observe and predict the input-output relationship at stage boundaries.

We use the non-linear regression module `lm` that is part of the R statistical analysis tool. It uses the LevenbergMarquardt algorithm [1] for regression. We make predictions for stages by traversing down the plan from source stages (refer Algorithm 1). At each stage we learn one function per incoming edge for the size of the data transferred along that edge and a function that relates the input and output sizes of a stage. We estimate the size of data transferred along each edge by applying the corresponding function to the predicted output of the corresponding source stage. The predicted input size of a stage is the sum of such sizes. We predict the output size of a stage by applying the learnt input-output function at the predicted input. In addition to data sizes *PerfOrator* also uses regression to estimate the CPU cycles spent on the computation within a stage.

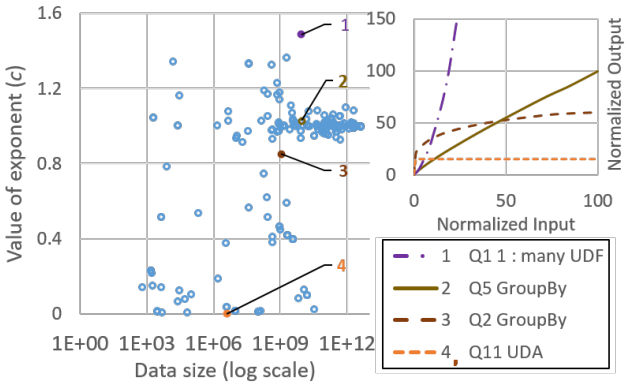


Figure 4. Input-output relationships: (on the left) values of c in $output = b(input)^c$ for all stages of BigBench queries; (on the right) input-to-output behavior for four selected operators.

4.1.1 Analysis of operators

Figure 4 shows the function forms observed in practice. It plots the function exponent against the output size on a log scale. While there are many exponents near 1 a majority of the functions are non-linear (validating our choice of functions). On close inspection we find that even within the stages of a single query non-trivial sub-linear, super-linear and near-constant relationships are seen in arbitrary order, and across queries similar operators exhibit very different behavior.

Our observations are consistent with function forms relational operators and UDFs are expected to exhibit (from analysis of cardinality estimation literature and manual inspection of UDFs respectively). We summarize the expected behavior of operators below.

Selection, projection and simple 1:1 UDFs Most unary relational operators (selections and projections) are expected to exhibit a linear behavior with truly random samples of the input to them [17], $c = 1, b \leq 1$. Sampling based techniques are known to estimate the selectivity (ratio of the output to the input) of such operators well. In practice we find that while selectivity can be estimated with high accuracy the estimated output cardinalities can still be off as the size of the input to the operator itself could be hard to estimate. Simple 1:1 UDFs can produce at-most one output record per input record and can be expected to have a similar behavior.

Grouping Aggregations and UDAs SQL comes with a fixed set of aggregations that all only emit a single output per group. The output cardinality of an aggregation depends on the number distinct values that the grouping key can take and distinct value estimators have been designed [7, 16] to

estimate their cardinality. UDA's and custom reducers generalize grouping aggregations by allowing users to write arbitrary aggregation functions that can produce zero or more rows per group. DVE does not directly apply to UDA's as many times the output cardinality of the aggregation is not proportional to the number of distinct values (The UDA of Q1 is an example). In section 2 we saw how using DVE for UDA's and multi-stage queries leads to poor estimates.

We find that while grouping SQL aggregations are sub-linear or linear $c \leq 1$, UDA's can be super-linear as well $c > 0$. Sometimes there are very few groups and the relationship appears like a constant function $c = 0$.

Joins and 1:many UDFs Natural joins are richly studied in literature. It is impossible to cover all literature here. Often joins are performed on a primary key. For such joins typically $c = 1$. For other natural joins we expect the relationship to be super-linear ($c > 1$) [17].

Other forms of joins like semi-joins and outer-joins are less common but do occur in practice. Notice that while joins are binary operators, our function directly relates the input size (sum of the sizes of the two tables) to the size of the join output. Other forms of functions could be explored but our evaluation reveals that this simple function works well in practice. 1:many UDFs can produce zero or more output records for every input record, they are expected have $c > 1$.

Count(*) and limit The last few stages often use operators like count(*) and limit. The size of the output here is typically a constant b, independent of the size of the input $c = 0$.

In summary we find that input-output relationships can be super-linear, sub-linear, linear or constant. The function form $output = b(input)^c$ can fit all these shapes.

4.2 The sampling step

The key challenge with sampling is to generate samples that (i) can flow through the entire query (ii) can be generated at a low overhead. Early database cardinality estimators build on uniform and independent samples of different tables [7, 16, 17]. [17] shows that sampling blocks of data instead of rows still produces unbiased estimates. A major drawback of sampling the tables independently is that join attribute values that appear rarely in one table can appear frequently in the other table and the estimate can be off if such values are not part of the sample. Literature shows that better sampling can be done if the possible joins relationships are known upfront (for example, referential integrity constraints for primay key based joins) or once the query is submitted [12, 23, 32, 36].

With big-data systems like Hive constraints between input tables are not explicitly specified. Also in our setting sampling online (once the query is available) can significantly increase the estimation overhead. In this paper we propose a simple practical sampling strategy that does not need explicit information about the schema of the data. We follow a skewed sampling strategy where we only create uniform random samples of large tables while we use complete small tables even during sample runs. This strategy is based on the observation [36] that joins are typically performed between, large fact tables that contain the information that needs to be analyzed, and small dimension tables that contain descriptive attributes for entries in the fact table. For such joins we expect to see some output when our skewed samples are joined, avoiding the sampling coverage problem. Further, the use of multiple samples to perform regression reduces the chances of poor coverage.

To amortize sampling overhead we reuse samples across queries and handle data modifications through triggers on Hive-DDL. Further we implement a block sampling strategy for distributed stores that allows us to sample data without scanning entire tables. Such stores split the data into multiple blocks and distribute them over the cluster. We implement block sampling by accessing the meta data about where the block are stored and choosing a subset of the blocks uniformly at random. We then collect the blocks in parallel.

4.3 The profiling step

We generate observations for regression through profile runs of the query on multiple samples of the input tables. The samples we use are typically small enough that we can run multiple profile runs concurrently on the same cluster. We create concurrent Hive sessions, that each run the query on one sample and collect statistics from the statistics server. The observations include the sizes of data at input and output of each stage, the size of data at source and target of each edge and CPU cycles spent in each stage. These observations are then fed into the regression step in Algorithm 1.

5. Performance estimation

Figure 5 shows our running example (BigBench Q1), flowing through *PerfOrator*. To recap, first a query plan is generated by a standard query optimizer. Next the *data-size estimator* annotates the plan with functions learned via non-linear regression. At this point the performance estimator kicks in (Algorithm 2). It parses the query plan top-down and does the following for each stage. First (§ 5.1), it leverages hardware profile to estimate how much time it would take for each task to read, write, process, and shuffle the amount of data estimated in the previous step. All estimated time components are Gaussian random variates with a mean and a variance. Next (§ 5.2), the system leverages an explicit model of framework parallelism to determine which time components are on the critical path and how much do they contribute to execution time. It also estimates the degree of parallelism for the stage by mimicking the algorithm used by the execution framework[3]. The result is a skyline of the parallelism and time taken for each stage. Other interesting metrics like overall execution time and dollar cost of running a query can be derived from the skyline.

[3] In the interest of space we do not describe the complete algorithm here.

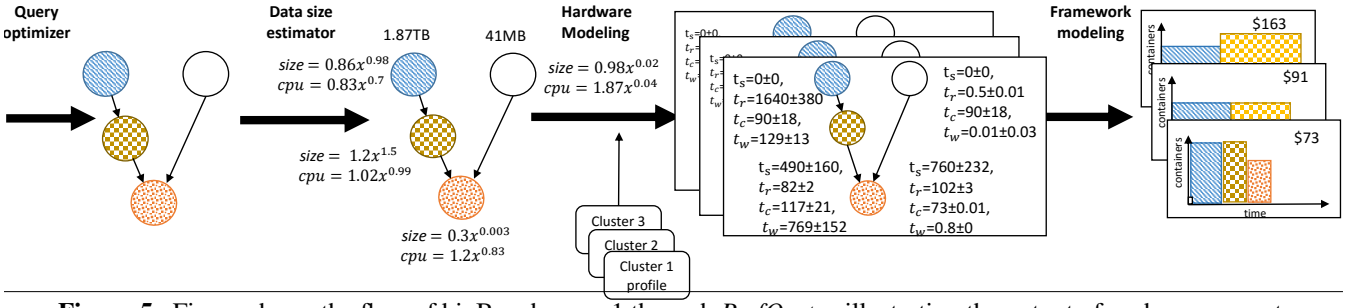


Figure 5. Figure shows the flow of bigBench query1 through *PerfOrator* illustrating the output of each component.

	Resources	Parameters	Query
Write	HDFS, Disk, AzureStore, in-mem store	*sizePerTask*, *rowSize*, *rCount*, *storeType*	Two stage query, the map stage uses a single task to produce one row per reducer and the reduce stage uses *rCount* reducers to write $\frac{sizePerTask}{rowSize}$ random rows of *rowSize* MB. The time spent in the second stage is recorded.
Read	HDFS, Disk, AzureStore, in-mem Store	*sizePerTask*, *rowSize*, *taskCount*, *storeType*	Single stage query that reads back appropriate data written by a write calibration query. The time spent in the stage is recorded.
Shuffle	network	*mCount*, *rCount*, *size*	Two stage query that transfers *size* MB of data from each of *mCount* mappers to *rCount* reducers. The time spent in the shuffle phase is recorded.
Compute	CPU	*loopCount*, *numTasks*	Single stage job that loops for *loopCount* iterations and performs a compute intensive hash function. The time spent in the stage is recorded.

Table 1. Calibration queries

Algorithm 2 PerformanceEstimator (annotatedPlan, cluster, framework)

```
1:  Profile = getClusterProfile(cluster)
2:  GaussianVar skyline, t*
3:  for stage in topDownTraversal(query) do
4:      tasks = NumberOfTasks(stage)
5:      for u in v.inputVertices do
6:          t_shuffle += SHUFFLE[u.outSize, u.tasks, v.tasks]
7:      end for
8:      t_read = READ[stage.inSize, tasks, stage.InStoreType]
9:      t_write = WRITE[stage.outSize, tasks, stage.OutStoreType]
10:     t_compute = COMPUTE[stage.cpuCycles, tasks, cluster,
        SampleRunCluster]

11:     skylinwe_height[stage] = Max(tasks,cluster.capacity)
12:     skyline_width[stage] = FrameworkModel(t*, framework)
13: end for
14: return skyline
```

5.1 Hardware Modeling

A major part of the hardware modeling is gathering performance profiles, this is query independent and done offline.

Cluster performance profiling via calibration queries Performance profiles are used to answer the following three questions about the performance characteristics of a cluster. 1) How much time would it take to read or write a given amount of data using a given number of tasks from a storage component? 2) How much time would it take to transfer a given amount of data over the network and store it at the target using a given number of source and target tasks? 3) How many compute cycles would it take to perform a fixed computation with a given number of tasks? *PerfOrator* runs Hive formulations of the calibration queries shown in table 1) to construct such profiles. We calibrate separately for MapReduce and Tez to capture the overheads of the framework when accessing the resources. We bound the calibration space by collecting profiles at a subset of data-size/task combinations and use linear interpolation for intermediate points. We vary the data size in powers of two starting with 1MB up to 20TB. And we vary the number of tasks from 1 to $20 \times cluster\_capacity$, where $cluster\_capacity$ is the maximum number of tasks that the cluster can run concurrently. We run each query multiple times (about 100 times) and fit a normal distribution to the observations across runs. Note that the profiles intrinsically capture the effects of data parallelism as they are parameterized by number of tasks/data-blocks to use. So while the estimates are per-task, they account for cross-task effects.

From data-sizes to time components The hardware profiles constructed above are used to translate the estimated data sizes and computational requirements (from § 4) into 4 time components per stage. Translation is made assuming

that the load is equally divided among all the tasks. Detecting skew and modeling it is a challenging problem that we do not address here. For each of shuffle, read, compute, write, the hardware profiles constructed offline are accessed with appropriate parameters (data size, number of tasks, storeType etc) and time estimate (both mean and variance) derived. As reads and writes could be performed, depending on the framework to main-memory, local disk, remote storage, appropriate hardware profiles are picked based on the execution plan. For shuffle, in case a stage has multiple predecessors we add-up the costs of shuffling data from each of them. For computation, we translate the estimated CPU-cycles (from Algorithm 1) on the sample run cluster to each target cluster. The translation is performed based on their relative compute performance as measured by the calibration queries. This way we estimate the performance on different clusters without having to profile online on each of them.

5.2 Framework Modeling

If an execution framework were to only exploit data parallelism then the execution time of a stage would simply be a sum of all the above time components for that stage (The analytical model proposed in Bazaar [22] is of this form). However, the execution framework typically exploits pipeline parallelism to overlap access to different hardware resources. When successive phases (shuffle, read, compute or write) are completely pipelined the time taken to complete them depends on the maximum of their time components.

Through a combination of log analysis and manual code inspection we construct offline parallelization summaries for two popular execution frameworks, namely Tez and MapReduce. The summaries identify what part of $t_{read}, t_{write}, t_{compute}, t_{shuffle}$ can run in parallel and from these execution time estimates are derived as below.

Tez The cost for each stage of Tez is as given below.

$$t_{stage_i} = t_{shufflePar} + t_{shuffleSeq} + t_{task}$$
$$t_{shufflePar} = Max\big(t_{shuffle} \times \frac{(w-1)}{w} - \frac{t_{prev}}{w}, 0\big)$$
$$t_{shuffleSeq} = \frac{t_{shuffle}}{w}$$
$$t_{task} = max(t_{read}, t_{compute}, t_{write})$$

Where:

$$prev = stage_k \; st. \; t_{stage_k} = max_{j \in Pred(Stage_i)}\{t_{stage_j}\}$$
$$w = \left\lceil \frac{tasks[stage_{prev}]}{cluster\_capacity} \right\rceil$$

Tez completely pipelines read, compute and write phases so the time to complete these phases is just the max of their costs. This is captured by the third term (t_{task}). Shuffle is more interesting, shuffle is typically overlapped with the last wave of tasks (a wave consists of *cluster_capacity* tasks) for the previous stage[4]. We model shuffle with two terms, the first term accounts for the parallelism between the shuffle of output produced by all but the last wave of tasks and the execution of the last wave. The second term accounts for the shuffle of the last wave of tasks, which happens sequentially.

Hadoop MapReduce Due to heavy use of disks and inefficiencies in implementation (as pointed out in [37] as well) MapReduce is unable to pipeline read and write phases. The model for MapReduce therefore has an additive term for writes. With MapReduce the DAG contains alternating map and reduce stages. Reduce stages shuffle data in from the previous map stage. Maps read in their data from HDFS (where reducers write) and do not have a shuffle phase.

For map stages:

$$t_{stage} = t_{readcompute} + t_{write}$$
$$t_{readCompute} = max(t_{read}, t_{compute})$$

For reduce stages:

$$t_{stage_i} = t_{shufflePar} + t_{shuffleSeq} + t_{readCompute} + t_{write}$$
$$t_{shufflePar} = max\big(t_{shuffle} \times \frac{(w-1)}{w} - \frac{t_{prev}}{w}, 0\big)$$
$$t_{shufflePar} = \frac{t_{shuffle}}{w}$$
$$t_{readCompute} = max(t_{read}, t_{compute})$$

Note that MapReduce ends up accessing more latency intensive resources and this can further slow down execution. This will be captured by the calibration queries.

Solving analytical equations

As described above, given time components for a query, the appropriate analytical equations are setup based on the parallelism summary for the execution framework. Note that solving the equations is trivial if the time components were point estimates. For *PerfOrator* the components are Gaussian random variables and we want to estimate a distribution for the stage time. The sum of two Gaussian distributions is also a Gaussian distribution $N(\mu_1, \sigma_1) + N(\mu_2, \sigma_2) = N(\mu_1 + \mu_2, \sigma_1 + \sigma_2)$. The max of two Gaussians does not have a closed form solution. We use a sampling based to approach to fit a Gaussian over a max of two Gaussians. We construct many sample pairs by sampling each Input Gaussian independently. We construct an output set which consists of the max value for each pair. A reasonable approximation for the output distribution is a Gaussian with mean and variance of the output set.

5.3 Discussion

The performance estimator captures the effects of pipeline and data parallelism at a high-level. It also captures the effects of memory optimizations employed by modern frameworks. It took us about 2 man months to extend the model from MapReduce to Tez. Other frameworks (Cosmos [6], Spark [37]) employ similar optimizations and we believe our models can be extended to them with a similar effort.

[4] If there are multiple predecessor stages we consider the overlap with the execution of the slowest of such stages (*prev* in equations).

6. Resource Optimizers

Cloud Provisioning This resource optimizer explores the space of different cluster provisioning options offered by the cloud provider (Azure in our case) and provisions the optimal cluster. The optimality criteria is defined by any objective function of completion time and cost incurred. The search space size is $O(n*k*t)$ where n is the maximum size of the cluster, k is the number of VM types available and t is the maximum number of time intervals considered for provisioning. In a typical setting, n can range from 1 to few thousands, k ranges from 10 to 50, and since time granularity is in the order of hours t is typically between 0 and few tens. The optimizer explores the space exhaustively along (n,k) dimensions by invoking *PerfOrator* on each (n,k) point and obtaining an estimated $\hat{t}$ per point. From this it can find the point that is optimal with respect to the objective function. After profiling, each invocation of *PerfOrator* is rather cheap (0.35 milliseconds). During experiments resource provisioning decision took 1.3 secs on average and never more than 4 secs. Trivial further speed ups can be achieved via parallel exploration.

YARN resource reservation A key challenge in shared BigData clusters is to offer Service Level Agreements(SLA) for production jobs, while keeping clusters highly utilized [10]. Cluster resource managers such as Hadoop/YARN provide users with the ability to "reserve" resources for production jobs. This consists in specifying ahead of job execution a skyline of resource needs. The resource manager accepting such reservation requests will ensure dedicated access to reserved resources, thus guaranteeing execution SLAs.

The key challenge here is to find the "tightest" skyline that can ensure that the SLA will be met. *PerfOrator* is a natural fit for this. Further, its ability to estimate per-stage average *and* variance is key to make principled decisions between safely over-estimating the job's need (but possibly wasting resources), and under-estimating the resource needs thus violating SLAs (but allowing better cluster utilization).

Practical issues such as stragglers, make handling this trade-off very difficult. The resource optimizer employs a clever trick to handle this. Instead of allocating rectangles of allocation, it produces for each stage an "L" shaped allocation, where lots of parallelism is given for the predicted time, but a "tail" is also allocated to account for stragglers (see Figure 7 for an illustration). This significantly mitigates SLA violation due to stragglers, with a low resource wastage.

7. Evaluation

Workloads We evaluate *PerfOrator* on: 1) BigBench [15], a benchmark suite for BigData systems currently in consideration for TPC standardization, 26 queries on a 10TB dataset, and 2) several production queries analyzing Bing logs, on a dataset of 7.5TB.

Hardware and framework settings Evaluation is performed on multiple Azure clusters, comprised of 3 different VM types (see Table 2), with different bandwidth to storage (not published), and in different data-centers (Central US, East US, South East Asia). We also evaluate on a production grade on-prem cluster with 80 nodes each equipped with 32 CPU cores, 12 HDD, with a 10GBs flat network. We evaluate queries on both the execution frameworks we model.

VM Type	cores	RAM	Disk	Price
D3	4	14GB	200GB	0.62 USD/Hr
D4	8	28GB	400GB	1.24 USD/Hr
D12	4	28GB	200GB	0.76 USD/Hr

Table 2. VM types

Comparisons We compare against optimal allocations whenever known, and *spotAdapt* a baseline estimator from past literature [25]. The baseline, proposed for single node query evaluation, employs regression to directly estimate execution time of each stage (as opposed to cardinality) on different AWS VMs. Just like *PerfOrator* it uses observations from sample runs as input to regression. Unless otherwise mentioned we use samples at 1,2 and 3% for both *PerfOrator*and the baseline. The profiling overhead of the baseline is the same as that of *PerfOrator*.

***PerfOrator* variants** In order to show the different contributions of our data-size estimator and performance estimator, we report the performance of *PerfOrator* using different data-size estimators: 1) our estimator from § 4 (simply called PerfOrator), 2) an oracle providing perfect data-size information (called OracleDS+PerfOrator in figures), and 3) the estimator used in the Hive optimizer (Hive-DS+PerfOrator). Note that to enable this scenario we allow the Hive optimizer to create statistics on data. This implies the data cannot be operated upon for a few hours after it has been bulk loaded. However, this option does avoid profiling and represents our proposed solution for scenarios in which even the limited profiling cost of our data-size estimator (analyzed in § 7.2.1), is not acceptable.

7.1 Resource optimization results

We begin by presenting results for the resource optimization scenario's of § 6. Later we provide a breakdown of the results for each estimator. To account for variance across runs we run each query at-least 5 times for each cluster configuration.

Resource provisioning We demonstrate the utility of *PerfOrator* towards picking the best (cheapest) cloud-based cluster configuration for each query in BigBench, this is akin to our motivational user-experiments in § 1. We consider six Azure clusters, made up of 3 VM types (D3, D12 and D4) and either 32 or 128 VMs each. Clusters are spread across three different data centers with different storage bandwidths, to further differentiate the options (hence stressing our hardware profiling models). *PerfOrator* observes a profile run for each query in BigBench on *one* cluster type/size, and estimates the performance across all cluster options, and then our cloud provisioner picks the cheapest option.

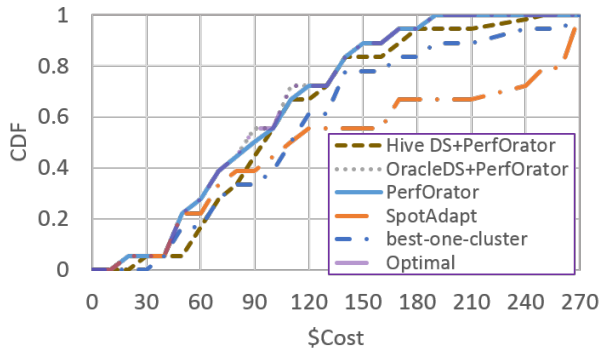


Figure 6. Cloud provisioning for BigBench.

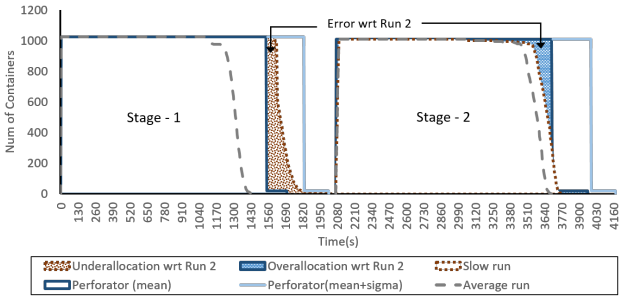


Figure 7. Actual and predicted skylines for Q1.

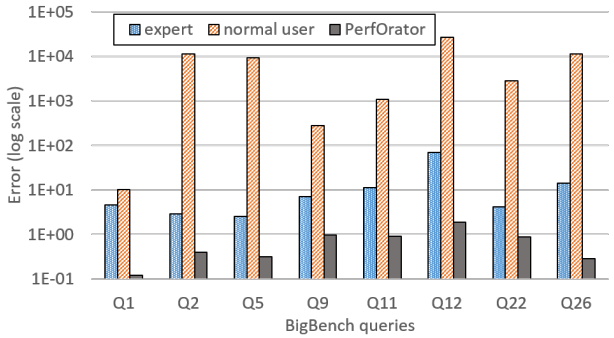


Figure 8. PerfOrator vs Humans: skyline prediction.

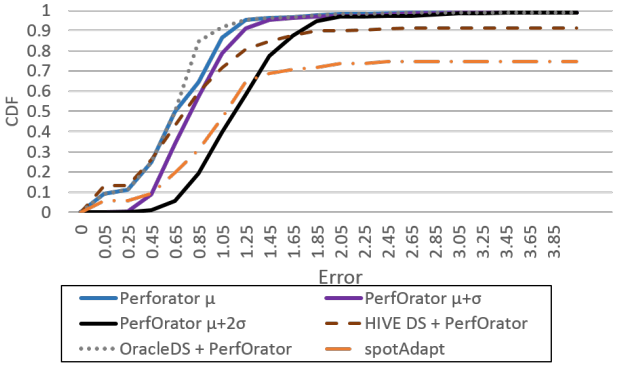


Figure 9. CDF of skyline area error per query run (BigBench + Bing).

Figure 6, shows the CDF of the costs incurred by all the queries under different allocations strategies. *PerfOrator* perfectly matches the optimal strategy for all but a couple of queries, for which it picks the second-best option (which is very close to optimal). The best-one-cluster solution is an optimal strategy if only one cluster can be chosen for all queries. While this looks good for most queries, the average cost increase over *PerfOrator*'s pick is 45% (due to few really poorly matched queries). Finally the baseline performs poorly, a regression on execution time does not seem sufficient to rank different cluster alternatives. A positive surprise for us, was how well Hive-DS+PerfOrator behaved. This strategy is our fallback in case online profiling is deemed too expensive. *Discussion* The reason Hive's results are so good is that this scenario is a "relative" comparison of allocation plans, much like what happens in query optimization where multiple options for the *same* query are compared. However, using the same strategy for allocating an entire workload (where comparisons across queries matter) leads to substantially worse results. Hive-DS+PerfOrator's errors grow by over 25%, while *PerfOrator* has the same accuracy because it is better suited to compare across queries.

Resource reservation (for job SLAs)

For this application *PerfOrator* estimatess skylines to reserve resources on a YARN cluster [10]. This is by far the hardest challenge for *PerfOrator* and for the human experts.

Figure 7 showcases the skyline reservations we generate (using *PerfOrator*'s mean and sigma estimates for each stage) for BigBench Q1 on a cluster capable of 1024 parallel containers. The core challenge is the trade-off between over- and under-allocation—which in turn affect SLA attainment and cluster utilization (overallocation is safe but wasteful).

In order to put our results in perspective, we compare (on a 8-query subset of BigBench) *PerfOrator* against two users (one expert, and one regular user), who were given the chance to run several profile runs, and inspect the data (for a substantial amount of time). The results are shown in Figure 8. To account for both over and under allocation, we calculate area error as $\frac{\sum|predicted-actual|}{\sum actual}$ (see Figure 7 for an illustration). *PerfOrator*'s error is typically at least an order of magnitude less than the one incurred by the users. Note that while the expert user more diligently profiled each run and manually interpolated the results, the "normal user", tried to generalize across queries, relying on their intuition of what the queries would do. Both incur very high errors, this confirms that skyline prediction is one of the hardest challenges in Resource Optimization.

Figure 9 summarizes the accuracy of skyline reservations in terms of an error CDF. We treat each run of the query separately, as we are interested in reserving sufficient resources even for slow runs. As can be seen *PerfOrator* does very

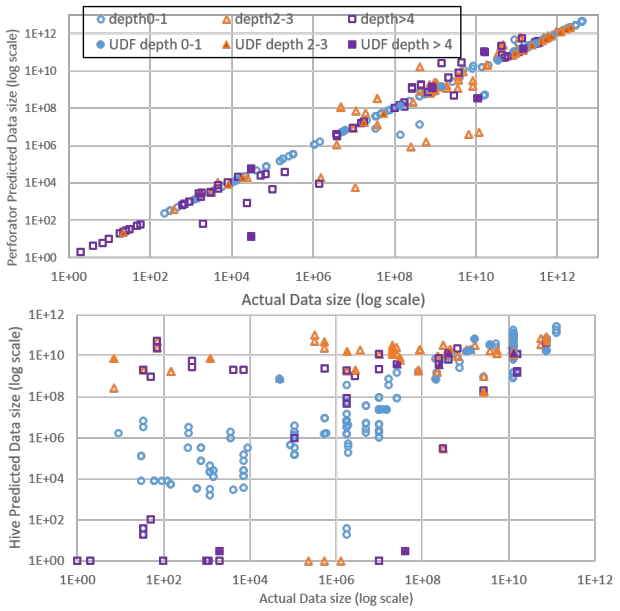


Figure 10. Accuracy of data-size estimation for *PerfOrator* $S\langle 1,2,3\rangle$ (above) and Hive (below). The plot is in log-log scale. Different colors are used to capture the depth of a stage within the query execution plan.

well, 90+% of runs have an error of less than $2\times$. Baseline trails significantly. Allocating at $\mu+\sigma$ does not increase the area error by much, and lowers the risk of SLA violation to $< 5\%$. Allocating at $\mu+2\sigma$ further lowers the SLA violation risk, but substantially increases the error. In this context, the Hive data-size estimator is not sufficiently precise and error remains very large for the top 10% of queries.

7.2 Analysis of estimators

7.2.1 *Data size estimation*

To examine the accuracy and overhead of our regression based data-size estimator we experimented with profile runs for several sample combinations. We create samples at 10 different sample rates of the input data (0.1%, 0.2%, 0.3%, 1%, 2%, 3%, 4%, 5%, 10%, and 15%). We periodically regenerated samples to make sure that our results are not biased by specific samples. We gathered accuracy and overhead results for many different combinations of the sample rates. We use the shorthand $S\langle x_1,x_2...x_n\rangle$ to represent estimations using sample runs at x_1 to x_n percent. For example $S\langle 1,2,3\rangle$ represents estimations made by using sampling rates 1%,2%,3% in algorithm 1.

Estimation Accuracy Figure 10 compares the actual data sizes with our estimates and Hive optimizer estimates on a log-log plot. *PerfOrator*($S\langle 1,2,3\rangle$) is very accurate with 97% estimates within the 10x error margin and about 90% of the estimates within the 2x margin. This is a significant improvement over cardinality estimates currently used by the optimizer in Hive where 41% of the estimates are off

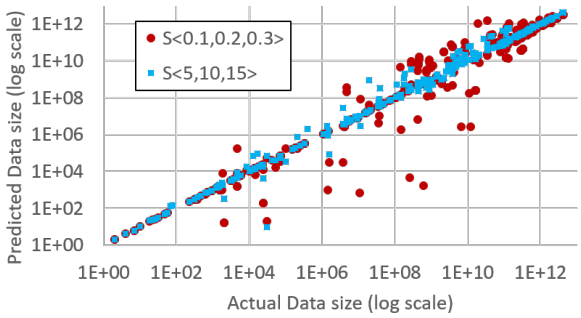


Figure 11. Accuracy of *PerfOrator* with profile runs at different sample rate combinations. The plot shows both the actual and predicted data-sizes for $S\langle 0.1,0.2,0.3\rangle$, $S\langle 5,10,15\rangle$. The plot is in log-log scale

by more than 10x and close to 65% are off by more than 2x. The estimates are as good for stages with UDFs as for other stages. Also unlike Hive where the accuracy drops significantly with increase in query depth, the degradation in accuracy is more gradual in *PerfOrator*.

Figure 11 shows the results for smaller/larger samples sizes. Interestingly, even $S\langle 0.1,0.2,0.3\rangle$ achieves good estimations with 82% estimates within a factor of 2. Unsurprisingly larger samples, such as $S\langle 5,10,15\rangle$ achieve better estimation, but the benefits are not likely worth the increased overhead (discussed next). We find $S\langle 1,2,3\rangle$ to be a very good practical compromise.

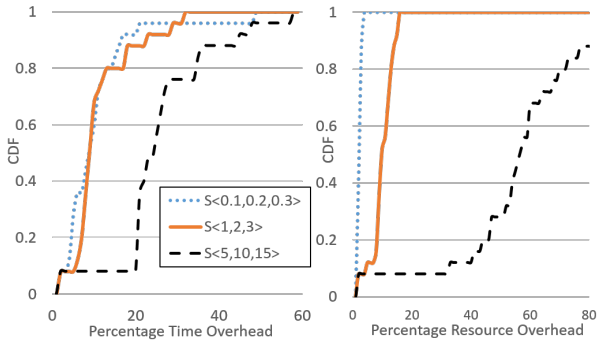


Figure 12. Estimation overhead as a fraction of execution time and cost.

Estimation overhead Figures 12 compares the estimation overheads in terms of time and resource consumption. The overheads are measured as the fraction of execution time and cost respectively of running the actual query. The plots shows CDFs for three different observations. As can be seen $S\langle 0.1,0.2,0.3\rangle$ has negligible overheads. In fact the time overheads are comparable to the overhead of the query optimizer in Hive today. $S\langle 1,2,3\rangle$ also has limited overhead (9% of runtime and 6% of resources in average), while $S\langle 5,10,15\rangle$ has substantial overhead for many queries.

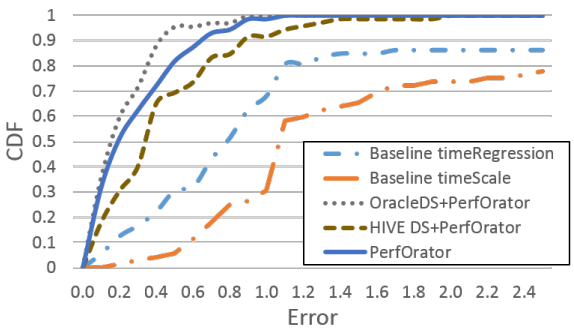


Figure 13. CDF of execution time estimation errors.

We also measure the overhead of sampling the tables at different sampling rates compared to full scans, but given we perform block-level sampling and cache samples across queries, this cost can be in first approximation ignored.

7.2.2 *Performance prediction*

Figure 13 shows the CDF of runtime estimation error for *PerfOrator* (in 3 variants), and two variants of the baseline. In addition to using 3 samples at 1,2 and 3%, we also evaluate *spotAdapt* when using one more observation at 5% (spotAdapt4samples in Figure). We plot the estimates for both execution frameworks in one CDF as both have very similar trends. As can be seen *PerfOrator* makes rather accurate estimates. Most importantly, we note that these errors cannot be lowered much further, as on average the actual execution time has a standard deviation of about 20%. The improvement over the baselines is very substantial. While adding an additional point improves the accuray the errors with the baseline remain very high. Similar to what we observed for cost estimation, Hive-DS+PerfOrator achieves reasonable runtime estimations. This is a good option in settings where online profiling is not possible.

8. Related Work

The overall problem we target, Resource Optimization, is rather novel, and to the best of our knowledge no system addresses it in its entirety. Ernest [33] comes closest, it proposes black-box models to pick the best cloud-based cluster for a job. Like *PerfOrator*, Ernest relies on profile runs, but there are a few important differences: Ernest does not implicitly generalize across hardware types, and it requires profile runs of the job on each VM-type; non-linear operators are supported, but through a manual process; most importantly, Ernest does not derive complete skylines of resources.

Query optimization Cardinality estimation is a well-studied subproblem of classical query costing. *Statistics-based techniques* collect and propagate statistics about data through operators [9, 21]. Propagation is sound under simplifying assumptions that may not hold in practice and sometimes lead to poor estimates [29]. High-velocity and unstructured data make it even harder to apply these techniques in our setting. *Sampling based techniques*, focus on select-project-join + aggregation queries, and leverage sampling and operator-specific models [7, 8, 17, 18]. These techniques do not generalize arbitrary UDFs. Also deep queries pose a challenge.

BigData performance estimation Prior work has focused on simpler variants of the problem we address. Bazaar [22] employs white-box models for MapReduce jobs and derives relative costing of options—we focus on wallclock time estimates and a broader class of application frameworks. Spot adapt [25] estimates the execution time and cost of queries running on a single VM instance. It employs linear regression to extrapolate the execution time from runs on samples of the data on a few VM options. While this works well for single node execution, our evaluation reveals that this approach does not capture the effects of parallel execution well, leading to significant mis-predictions. Packing light [11] predicts the throughput of a workload on the cloud based on machine learning. However, the estimator requires full runs on a testbed, making it prohibitively expensive in BigData settings. Detailed white box model to predict the execution time of MapReduce jobs have been proposed in [27, 35]. The models are closely tied to the specific implementation details of MapReduce and do not easily translate to other frameworks. Other black-box techniques [13] have been used to estimate performance metrics for purely relational queries (no UDFs) running with fixed hardware. The accuracy of their technique depends on precise cardinality estimates. DYNO [24] proposes a profile driven optimizer to dynamically optimize BigData queries with UDFs. Finally, morpheus [4] a parallel effort, estimates full skylines for recurring production jobs. With *PerfOrator* we tackle a more challenging variant of these problems.

9. Conclusions

Hardware has been traditionally considered a "constant" in query optimization. Large cluster and pay-as-you-go clouds create a new non-trivial problem for BigData workloads: Resource Optimization. This consists of finding the best hardware resources for a given execution plan. We demonstrated that this problem is far from trivial and that solving it can have substantial impact on query performance and cost.

Our main contribution is a novel approach to model resource-to-performance behavior of arbitrary BigData queries, which we package in a system called *PerfOrator*. *PerfOrator* is capable of predicting the overall resource skyline of a query, thus also cost and runtime, across hardware types (without the need to profile the query on each). *PerfOrator* employs non-linear regression, hardware calibration, and analytical modeling to handle high-velocity unstructured data, UDFs, and heterogeneous hardware. *PerfOrator* is integrated with both the Azure cloud and Hadoop/YARN. Validation against production workloads as well as standard benchmarks demonstrates *PerfOrator*'s practicality.

References

[1] https://en.wikipedia.org/wiki/Levenberg-Marquardt_algorithm.

[2] Apache Calcite. http://calcite.apache.org/.

[3] Azure vm pricing. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/, 2016.

[4] S. Abdu Jyothi et. al. Morpheus : Towards automated slas for enterprise clusters. OSDI'16.

[5] S. Agarwal, H. Milner, A. Kleiner, A. Talwalkar, M. Jordan, S. Madden, B. Mozafari, and I. Stoica. Knowing when you're wrong: Building fast and reliable approximate query processing systems. SIGMOD '14.

[6] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. OSDI'14.

[7] M. Charikar, S. Chaudhuri, R. Motwani, and V. Narasayya. Towards estimation error guarantees for distinct values. PODS '00.

[8] S. Chaudhuri, G. Das, and U. Srivastava. Effective use of block-level sampling in statistics estimation. SIGMOD '04.

[9] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Found. Trends databases*, 2012.

[10] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao. Reservation-based scheduling: If you're late don't blame us! SoCC '14.

[11] J. Duggan, Y. Chi, H. Hacigm, S. Zhu, and U. etintemel. Packing light: Portable workload performance prediction for the cloud. ICDEW'13.

[12] C. Estan and J. F. Naughton. End-biased samples for join cardinality estimation. ICDE '06.

[13] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. ICDE '09.

[14] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall Press, 2 edition, 2008.

[15] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H.-A. Jacobsen. Bigbench: Towards an industry standard benchmark for big data analytics. SIGMOD '13.

[16] P. J. Haas, J. F. Naughton, S. Seshadri, and L. Stokes. Sampling-based estimation of the number of distinct values of an attribute. VLDB '95.

[17] P. J. Haas, J. F. Naughton, S. Seshadri, and A. N. Swami. Selectivity and cost estimation for joins based on random sampling. *J. Comput. Syst. Sci. June 1996*.

[18] H. Hacigumus, Y. Chi, W. Wu, S. Zhu, J. Tatemura, and J. F. Naughton. Predicting query execution time: Are optimizer cost models really unusable? ICDE '13.

[19] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: a platform for fine-grained resource sharing in the data center. NSDI'11.

[20] A. Hulgeri and S. Sudarshan. Parametric query optimization for linear and piecewise linear cost functions. VLDB '02.

[21] Y. Ioannidis. The history of histograms (abridged). VLDB '03.

[22] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Bridging the tenant-provider gap in cloud services. SoCC '12.

[23] S. Kandula, A. Shanbhag, A. Vitorovic, M. Olma, R. Grandl, S. Chaudhuri, and B. Ding. Quickr: Lazily approximating complex adhoc queries in bigdata clusters. SIGMOD '16.

[24] K. Karanasos, A. Balmin, M. Kutsch, F. Ozcan, V. Ercegovac, C. Xia, and J. Jackson. Dynamically optimizing queries over large scale data platforms. SIGMOD '14.

[25] D. Kaulakienė, C. Thomsen, T. B. Pedersen, U. Çetintemel, and T. Kraska. Spotadapt: Spot-aware (re-)deployment of analytical processing tasks on amazon ec2. DOLAP '15.

[26] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *VLDB 2015*.

[27] H. Lim, H. Herodotou, and S. Babu. Stubby: A transformation-based optimizer for mapreduce workflows. *VLDB 2012*.

[28] C. H. Papadimitriou and M. Yannakakis. Multiobjective query optimization. PODS '01.

[29] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. Leo - db2's learning optimizer. VLDB '01.

[30] I. Trummer and C. Koch. Multi-objective parametric query optimization.

[31] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. SOCC '13.

[32] D. Vengerov, A. C. Menck, M. Zait, and S. P. Chakkappen. Join size estimation subject to filter conditions. *VLDB 2015*.

[33] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. NSDI, 2016.

[34] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. EuroSys 2015.

[35] Y. Wang, Y. Xu, Y. Liu, J. Chen, and S. Hu. Qmapper for smart grid: Migrating sql-based application to hive. SIGMOD '15.

[36] F. Yu, W.-C. Hou, C. Luo, D. Che, and M. Zhu. Cs2: A new database synopsis for query estimation. SIGMOD '13.

[37] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. HotCloud'10.