Automatic Tuning of SQL-On-Hadoop Engines on Cloud Platforms

Prasad M Deshpande
Qubole India
Bengaluru, Karnataka, India
prasadmd@acm.org

Amogh Margoor
Qubole India
Bengaluru, Karnataka, India
amoghm@qubole.com

Rajat Venkatesh
Qubole India
Bengaluru, Karnataka, India
rvenkatesh@qubole.com

ABSTRACT

More and more companies are running Big Data workloads on cloud platforms. Configuration tuning of these data engines continues to be an essential but difficult undertaking. Cloud platforms further add to this complexity due to elasticity of compute resources and availability of different machine types. Data engineers have to choose the correct machine type as well as the number of machines along with the other configuration options.

In this paper, we address the problem of automatically determining good configuration parameters for SQL workloads on SQL-On-Hadoop Engines like Hive and Spark, with the goal of optimizing resource usage and cost. We chose to focus on SQL workloads because SQL or SQL-like languages continue to be the most popular choice of ETL engineers and analysts. We present two alternatives – an iterative method based on repeatedly executing queries with different configuration parameters and a model based method based on simple rules and insights. We studied the effectiveness and practical usefulness of these methods on both synthetic and real workloads.

We show that very simple models of SQL Data Engines can provide very good recommendations compared to the default configuration. In fact, we found that these were often better than those chosen by experts on real customer workloads. The principles behind our model based approach are generic and can be adopted for any big data engine.

CCS CONCEPTS

• **Information systems → MapReduce-based systems**; **Relational parallel and distributed DBMSs**; **Autonomous database administration**;

KEYWORDS

Big Data, SQL, Hadoop, Hive, Presto, Spark, Automatic Performance tuning

ACM Reference Format:
Prasad M Deshpande, Amogh Margoor, and Rajat Venkatesh. 2018. Automatic Tuning of SQL-On-Hadoop Engines on Cloud Platforms. In *Proceedings of ACM SIGMOD conference, Houston , Texas USA, June 2018 (SIGMOD 2018),* 12 pages.
https://doi.org/10.475/123_4

1 INTRODUCTION

With the widespread adoption of cloud technologies, companies of all sizes are choosing to host their compute infrastructure on cloud platforms like AWS, Micrsoft Azure and Google Cloud Platform. Many of the best practices from in-house data center administration and tuning are also carried over to the cloud. However, cloud platforms have several unique properties like elasticity, separation of compute and storage, short lived clusters, and availability of different machine types. As big data infrastructure companies have matured, they are building systems that exploit some features of cloud platforms effectively. For example, Qubole provides auto-scaling and heterogenous clusters for Apache Hadoop, Apache Spark and PrestoDb systems.

However, automatic workload tuning and capacity planning have not yet reached the same level of maturity as auto-scaling on cloud platforms. By tuning a workload, we mean determining a good set of configuration parameters and cluster settings for the queries to run efficiently. Organizations need to rely on experts to optimize big data systems for their workloads. However, the complexity of big data technologies and workloads combined with the choice provided by the flexibility of cloud platforms makes manual tuning unscalable.

In this paper, we address the problem of automatically determining good configuration parameters for SQL workloads with the goal of optimizing resource usage and cost. We present two alternatives – an iterative method based on repeatedly executing queries with different configuration parameters and a model based method that relies on simple rules and insights. Both of these approaches have been proposed in some form in prior works. For example, [9] uses an simultaneous perturbation stochastic approximation (SPSA) based iterative algorithm to discover good configuration parameters. In the model based category, Starfish [4, 6] was the one of the first systems to develop a comprehensive model for Hadoop map-reduce (MR) based systems. These previous works were directed towards generic workloads running on Hadoop systems, which increases the complexity of the problem due to the large number of possible configuration parameters and factors affecting system performance. This makes the iterative approach too expensive in practice due to large number of iterations. The model based approach, on the other hand, becomes very complex and brittle since they require the model to simulate every aspect of a data engine.

Our experience with customer workloads at Qubole suggests that SQL or SQL-like languages continue to be the most popular choice of ETL engineers and analysts. At Qubole, approximately 75% of the workloads are expressed in SQL or SQL-like languages. Thus, we chose to focus on SQL Workloads in our work rather than generic workloads. Since the mechanism of SQL query processing is well understood, we can use that knowledge to identify a smaller subset of configuration parameters that have significant impact on the query processing time and focus on optimizing them.

We started with an iterative approach applying some more optimizations such as discretization, range reduction and dimension independence to further reduce the search space. Experiments on synthetic and real datasets show that the iterative algorithm was able to reduce resource utilization significantly (upto 80%) compared to the default (for synthetic workloads) or expert chosen values (for real workloads) in a small number of iterations (usually around 10-15). However, the dollar cost of even the smaller number of iterations makes it practically infeasible.

We then moved to the model based approach that uses rules and heuristics to optimize performance and cost. Rules for performance optimization use data statistics from the catalog or previous runs. We further extended the model for cloud platforms by adding rules that use machine type to factor in the resources available for a workload. The model optimizes cost by computing expected run times for all machine types and choosing the best combination of run time and price per machine. The goal of the model was not to predict the run time exactly, but rather to be able to accurately compare the relative performance of two sets of configuration parameters. The surprising result is that a very simple rule based model is able to provide very good recommendations leading to significant savings in resource usage and cost. In fact, we found that these were often better than those chosen by experts on real customer workloads.

The rest of the paper is organized as follows. Section 2 gives an overview of the related work and Section 3 defines the objectives of the optimization process. Section 4 describes the iterative approach and its results on real and synthetic datasets. Section 5 presents the insights that feed into developing the model based approach, which is detailed in Section 6 along with the experimental results. An extension of this approach for cloud platforms is presented in Section 7. We finally conclude in Section 8.

Experimental Setup. Qubole[8] provides proprietary versions of Apache Hive[15], Apache Spark[18] and Facebook Presto[7] to its customers. All experiments were run on Apache Hive and Apache Spark using Qubole Data Service (QDS) on Amazon Web Services (AWS). The models and experiments do not use any proprietary extensions of QDS. Due to budget and contract constraints, we have presented results from different customer workloads which consists of different data, queries, engine, cluster and machine configuration. To ensure consistency, we present results for each approach (iterative, model and cloud model) from a single company using a specific dataset, query set and data engine. Results for each approach are self-contained and independent. Moreover the variety of experiments and setup show the versatility of our approach.

The paper contains experimental results from a subset of TPC-DS[13] queries containing FILTER, GROUP BY, JOIN and SUBQUERIES. Cosmetic changes were made to run these queries on Apache Hive and Apache Spark. The paper also contains results of queries run on real workloads. These queries were provided by customers. Customer queries are from ETL workloads. These queries are of similar complexity to TPC-xBB benchmark[12] and include FILTER, GROUP BY, JOIN, SUBQUERIES and various types of UDFs.

A brief overview of the customers who provided permission to run experiments and their technology stack are:

- Customer 1: A large e-commerce company that uses Apache Hive executed on Hadoop2 MR engine to process click stream and product data.
- Customer 2: A large travel logistics company that uses Apache Hive executed on Hadoop2 MR engine to process data on demand and usage of its services.
- Customer 3: A Platform-As-A-Service company that uses Apache Spark to process telemetry data from its platform.

Section 4 on iterative approach contains experimental results at Customer 1 using Apache Hive on Hadoop 2 MR engine. For this reason we also present results of the synthetic workload using Apache Hive on Hadoop 2 MR engine. In Section 5 and Section 6 we generate insights and build a model using Apache Spark. We ran experiments for model-based approach at both Customer 2 and Customer 3. Therefore we modified the model for Apache Hive on Hadoop and present results for both. In Section 7 we present results from Customer 3 on Apache Spark only.

2 RELATED WORK

Most relational databases have auto-tuning tools for physical database design. Self-tuning Database Systems: A Decade of Progress[1] has a good survey of research and tools in this area. Most database systems have not put similar attention to determine best values of configuration parameters based on workloads. Traditionally effort has focused on specific class of parameters (e.g.[14]) or on ranking critical configuration parameters[2]. All these techniques are based on using the optimizer cost model to simulate database behavior. The disadvantage is that the cost model may not model the effects of all important parameters.

iTuned[3] is one of the first attempts to holistically tune configuration parameters in modern database systems. iTuned consists of a planner which plans experiments and an executor that run experiments to choose the best parameters and the best values for a specific workload. iTuned is also a good example of a system that ran experiments on user workloads and queries compared to using models to simulate data engines. Our preferred approach focuses on SQL-on-Hadoop engine and uses models to eliminate cost of experiments.

The popularity of Apache Hadoop ecosystem (Apache Hive, Apache Spark etc) increased interest in choosing configuration parameters automatically. The developers of these systems exposed engines parameters, documented them and encouraged users to change them based on their workload. The main motivation was that the type of workloads and data processed on these systems were varied and the default values were suboptimal. It was assumed that the administrators have to manipulate parameters for a successful installation. Similar to tools in database systems, there are two possible approaches - model based and execution based.

Starfish[6] was one of the first to model hadoop performance. Starfish consists of a *profiler, what-if* engine and a *cost-based model*[4]. The *profiler* collects statistics from previous runs of a customer workload such as bytes flowing through the system and time taken using dynamic instrumentation. The profiler provides a view of timings (wall clock timings of various phases), data flow (bytes input/output in every phase) and resources (CPU, networking, memory) used. The *what-if engine* simulates the behavior of the Map Reduce engine and can predict the effect of changing the value of a configuration parameter. The *cost based optimizer* uses the *what-if engine* and recursive random search (RSS) for tuning the parameters for a Hadoop job. The components of Starfish were extended with a new component - *Elastisizer*[5] - to automatically size clusters on cloud platforms.

Compared to Starfish which is applicable to all types of workloads on a MapReduce Engine, our approach differs in focussing on SQL workloads only. The focus on SQL workloads reduces the number of metrics required in profiler phase as well as the number of configuration parameters in the what-if engine. This approach has very practical benefits. An approach like Starfish has to consider every engine like Apache Hive, Apache Spark, Apache Impala and PrestoDb separately as the internal architectures are very different. By simplifying the model to a few common characteristics of all SQL engines, our approach can be extended to other SQL engines much more easily. For example, in this paper we provide models for both Apache Hive and Apache Spark. Moreover these projects move fast and the internal architectures change often in every release. The chances of requiring changes is lesser if the model is based on few parameters important for SQL workloads. The disadvantage of our approach is that it is not applicable to all types of workloads.

There are multiple approaches [17] [10] that use machine learning techniques to cluster jobs based on their profiles. Based on the profile, the most optimal cluster configuration is used. Aroma [10] optimizes resource allocation and cost of jobs. In the offline phase, using a training set, the jobs are clustered (using variants of k-means) according to their respective signatures. In the online phase, Aroma trains a SVM which makes accurate and fast prediction of a job's performance for various configuration parameters and input data sizes. The optimization function in Aroma considers the machine type, number of machines and unit cost of the machine types with a constraint on run time of the query. Our approach uses a rule based mathematical model instead of statistical or machine learning approaches.

CherryPick[11] and Sandeep Kumar et al.[9] represent an alternate school of thought. Mathematical or statistical models for complex data engines is hard to build. Moreover maintenance of these models is also hard as these data engines evolve. In this approach, experiments with alternate values of configuration parameters are executed instead of simulated in a statistical or mathematical model. The obvious disadvantage is that experiments cost money and the main goal of these approaches is to make every experiment count. In [9], the authors use SPSA and Bayesian Optimization in [11] to choose an optimal set of experiments to find the best values for a pre-determined set of configuration parameters. In Section 4 we tried an iterative execution approach and found it to be impractical in terms of dollar cost. At the same time it is hard to manage detailed models of complex engine and therefore we propose a simple model focused on important SQL operations.

3 OPTIMIZATION METHODOLOGY

We now present the optimization functions, parameters and assumptions to tune SQL-on-Hadoop data engines. Even though the model is specific to run time features of SQL-on-Hadoop engines, the principles can be translated to other engines.

3.1 Optimization Function

The optimization function should consider both cost and performance of a workload. In this section, we develop a metric to capture cost and performance of SQL-on-Hadoop engines running on Hadoop2 Yarn Containers. Hadoop divides machines into containers and each container is assigned a fixed amount of memory and CPUs. Workloads are submitted to Hadoop by Hive or Spark as a set of tasks which are scheduled by Hadoop on containers. The first metric we considered measures the total resource utilization. Since memory and CPU are both important resources, the cumulative resource utilization is given by:

$$\mathcal{R} = \sum_{i=1}^{n} t_i \times m_i \tag{1}$$

In the above equation, n is the number of tasks, t_i is the execution time for task i and m_i is the memory allocated to the container on which the task is scheduled. The time factor in the equation tracks the performance of the query. In a cloud deployment, the dollar cost of running the workload becomes an important metric. If the workload under consideration is the only one running on the cluster, then the cumulative time that the workload is scheduled on all the containers is a good proxy for the cost of the query, as given by:

$$\mathcal{T} = \sum_{i=1}^{n} t_i \tag{2}$$

Different machine types on a cloud platform can have different monetary costs depending on the number of cores, amount of memory, storage and network speeds. The actual dollar cost of a workload taking a cumulative time $\mathcal{T}_x$ on a cluster consisting of machines of type x having a rental rate of r_x per core, per unit time is given by:

$$C = \mathcal{T}_x \times r_x \tag{3}$$

3.2 Parameters

From OtterTune[16], a tool to auto tune Databases like MySQL and PostgreSQL, it was found that tuning only few knobs can improve performance significantly. This finding carries over to SQL-on-Hadoop engines as well. We chose a few critical parameters to optimize by consulting experts in the domain. These parameters can be set for each job separately and can thus be tuned for each query in the workload. Table 1 lists these job parameters for Mapreduce and Spark engine.

4 ITERATIVE METHOD

In the first approach, we actually execute each query multiple times with different configuration parameters to determine the optimal set of parameters. For each candidate set of configuration parameters, the query is run and the target metric, such as total resource usage, is measured. By comparing the metrics across different runs with

Parameter	Hive on MR	Spark
Mapper memory Reducer memory	mapreduce.map.memory.mb mapreduce.reduce.memory.mb	spark.executor.memory
Mapper parallelism	mapreduce.input.fileinputformat.split.maxsize mapreduce.input.fileinputformat.split.minsize	
Reducer parallelism	hive.exec.reducers.bytes.per.reducer	spark.sql.shuffle.partitions
Executor cores	Not applicable	spark.executor.cores
Mapper Buffer	mapreduce.task.io.sort.mb	Not applicable

Table 1: Parameters of the Job to be optimized

different configuration parameters, we can determine a good set of parameters to use.

4.1 Assumptions

The main challenge in such methods is to limit the number of trials, since each execution takes up resources and has a monetory cost associated with it. Earlier approaches based on iterative execution have used various techniques such as noisy gradient [9] to converge to a solution faster. In our method, we make use of domain knowledge and heuristics to reduce the search space. Specifically, we employed the following strategies to reduce the parameter space to be explored.

Parameter reduction: The search space is exponential in the number of parameters to be optimized. There are a large number of parameters that can be set for any query. As described in Section 3, we have identified a smaller set of parameters that have a relatively larger impact on the performance of sql queries. We restrict the search to these parameters, thus reducing the search space. These parameters have been listed in Tables 1. The search space with a restricted set of parameters is shown in Figure 1(b).

Discretization: Each parameter, such as memory or partition size, can take a large number of values. However, it can be observed that small changes in the parameters do not have a significant impact. Thus, instead of trying each possible value, it is sufficient to discretize the parameter range and consider only a subset of values for each parameter. These values are placed at a reasonable distance from each other so as to have a significant impact on the query performance. For example, instead of varying memory in units of 1 MB, we can vary it in multiples of 128 MB. The resulting search space is shown in Figure 1(c).

Range reduction: The range of values for each parameter is further restricted based on domain knowledge about what a good range for that parameter would be. The knowledge about a good range can be gained by either talking to experts or by looking at some other metrics. For example, for the Hive on MR engine, consider the mapper_time metric that measures the average time taken by a mapper. If the mapper time is too low, the overhead of starting the mappers is large compared to the actual work done by the mapper. Since the mapper time is inversely related to the number of mappers, the number of mappers need to be reduced. On the other hand, if the mapper time is large, then the job parallelism is restricted and the end to end clock time taken for the query will be high. In this case, more mappers are needed to reduce the work that each mapper has to do. A good acceptable range for this metric could be from 240s till 1800s. If a set of config params results in mapper_time beyond the acceptable range, it should not be considered in the search process. For example, mapper_time is affected by mapreduce.input.fileinputformat.split.maxsize and the correlation is direct, i.e. mapper_time increases as we increase mapreduce.input.fileinputformat.split.maxsize. Thus the split maxsize should be constrained to a range that will lead to a reasonable mapper_time. In our experiments, we restricted splitsize to between 128 MB and 1 GB. The resulting search space is shown in Figure 1(d).

Dimension independence: We make an assumption that the parameters are not correlated to each other. This enables us to optimize each parameter independently of the others. Thus, rather than exploring all the points in the search space, the algorithm explores only one set of values for each parameter as shown in Figure 1(e). This is a very strong assumption, which may not hold in practice. For example, the mapper memory (mapreduce.map.memory.mb) and the splitsize (mapreduce.input.fileinputformat.split.maxsize) are correlated, since more memory is needed by the mappers as the splitsize increases if spills are to be avoided. Even in this case, the algorithm will find the best value for memory after fixing the splitsize or the best splitsize after fixing the memory. So overall the configuration chosen will be a reasonably good one.

4.2 Algorithm

The overall method is listed in Algorithm 1 and is fairly straightforward. It starts with the default value for each configuration parameter (Line 1). It then iterates over the parameters and for each parameter it explores a range of values from low to high, varying it with a minimum step size (Lines 2–6). It runs the query with the chosen parameter values and measures the metric (such as running time or utilization). It finds the value for which the metric is optimized and fixes the value of the parameter to that value before moving on the next parameter (lines 7–12). Finally, it outputs the set of good parameter values V that are discovered in the process.

4.3 Results

We evaluated the effectiveness and practicality of the iterative method by running experiments on both synthetic and real workloads. The experiments were carried out for a Hive on MR engine. The results demonstrated the importance of optimizing the configuration parameters, but at the same time motivated us to explore the model based method instead of the iterative method. We thus

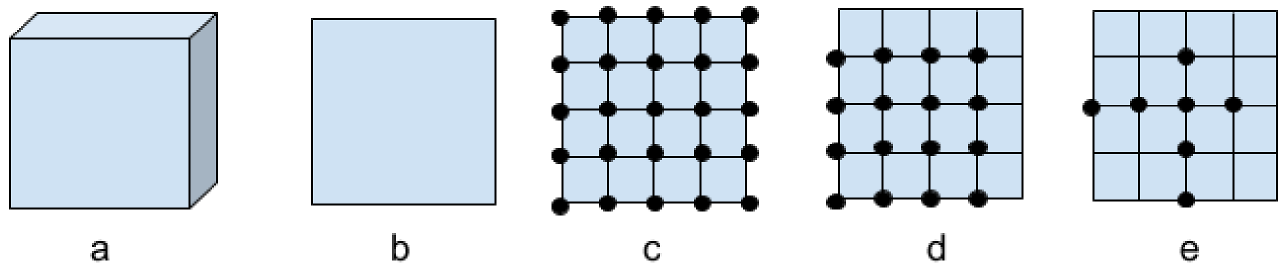

Figure 1: Reducing the search space

Algorithm 1 ***Iterative Search***

```
Input: Set $\mathcal{P} = \{p_1, p_2 \dots p_n\}$ of parameters to be determined, the metric $m$ to be optimized, the query $Q$
Output: The values for parameters in $\mathcal{P}$ that optimize $m$
 1: Let $V = \{v_1, v_2, \dots v_n\} = \{p_1^d, p_2^d, \dots p_n^d\}$ // $p_i^l, p_i^d, p_i^h$ and $p_i^s$ denote the low value, default value, high value and discrete step size for parameter $p_i$
 2: for Param $p_i$ in $\mathcal{P}$ do
 3:     $m_{best} \leftarrow null$
 4:     for Value $v$ from $p_i^l$ to $p_i^h$ in steps of $p_i^s$ do
 5:         Replace $v_i$ by $v$ in $V$
 6:         Run $Q$ with parameter setting $V$ and measure the metric $m$
 7:         if $m_{best}$ is $null$ or $m$ is better than $m_{best}$ then
 8:             $m_{best} \leftarrow m$
 9:             $v_{best} \leftarrow v$
10:         end if
11:     end for
12:     Replace $v_i$ by $v_{best}$ in $V$ // Best value for parameter $p_i$ is found and used in further search
13: end for
14: return $V$
```

present the results here before moving on to describing the model based approach.

Synthetic Workload. For synthetic workload, we used the TPC DS dataset (scale 1000) and queries. Each query had to be run a number of times to discover a good set of configuration parameters. To keep the time for the experiments reasonable, we had to use a somewhat small dataset. The average data read by each query was of the order of 20 to 40 GB. The experiments were run on a 4 node cluster on AWS with machine type r3.xlarge. The metric to be optimized was the total resource utilization (Equation 1) and parameters whose values are to be determined are the ones mentioned earlier in Table 1. For each query, we compare the best and worst configuration discovered with the default one in terms of the resource utilization. The results are shown in Figure 2. It plots the ratios of the resource usage of minimum to default, maximum to default and maximum to minimum. The min corresponds to the best configuration discovered and max corresponds to the worst configuration among the ones tried out. The graph shows that the min to default ratio varies from 0.18 to 0.73, which indicates that the iterative search leads to a configuration that can save upto 82% in resource usage over the default configuration. The ratio of max to default is mostly close to 1 and in some cases goes upto 1.80. This indicates that in many cases, the default configuration itself was the worst one (same as max). This is due to the fact that the default split size was 128 MB, leading to a large number of mappers, each processing small amount of data. Since there is some overhead (5-10s) in starting the JVM for each mapper, the startup time becomes significant percentage of the total mapper time if the mapper itself takes very less time to process the data. As a result, the overall query time suffers and resource utilization increases. This suggests that a different default with a larger split size would be more appropriate. Finally, the ratio of max to the min varies from 1.47 to 5.41, indicating that the configuration parameters do make a significant difference in query execution.

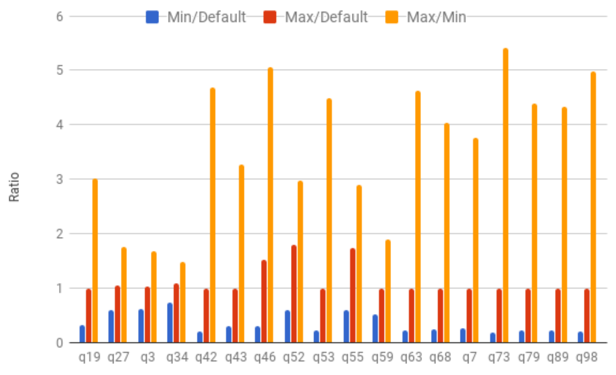

Figure 2: Iterative algorithm on TPC DS[1]

Real Workloads. We evaluated the iterative method over real customer data and queries (Customer 1). The workload consisted of three Hive queries running on Hadoop2 MR. These queries were very large and complex and part of their analytics workflow. The input data size was about 400 GB for two of the queries and 200 GB for the third query. The cluster consisted of a r3.2xlarge master node and 30 slave nodes of type r3.8xlarge on AWS Cloud. Figure 3 shows the percentage savings in the total resource utilization cost

(Equation 1) of the three queries with the configuration discovered by the iterative algorithm, compared to the cost with the configuration that the customer was using in production. The results show that the iterative algorithm was able to achieve significant savings upto 77% even for production workloads, indicating that many production workloads are not fully optimized. However, the iterative algorithm took over 50 hours and cost $5000 since the cluster was quite large with expensive machines.

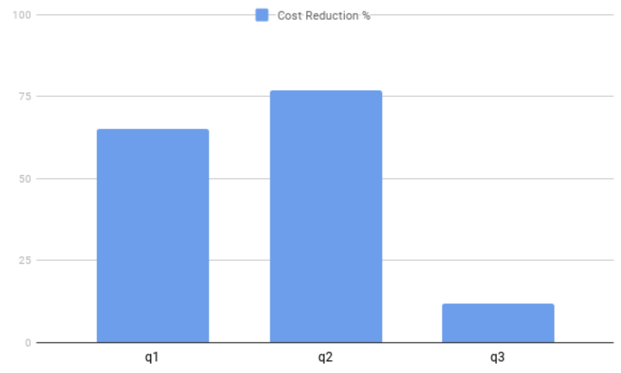

Figure 3: Iterative algorithm on real workload

Discussion. The results on synthetic and real datasets show that finding good configuration parameters can lead to significant savings in query costs. The iterative algorithm is able to discover good configuration parameters in a small number of iterations (usually around 10-15 iterations). However, there are some practical limitations as listed below:

- The dollar cost of the optimization process can be significant as seen in the real workload. In this case, the customer had 1000 more queries. It may be possible to make the search more efficient and reduce the number of iterations. Since customers have 100s or 1000s of queries, even 10 or 50 fold reduction is not sufficient to make the approach economical.
- For ETL queries, the approach requires shadow clusters and queries. The queries had to be reviewed multiple times to make sure production clusters and tables were not affected. The cost in terms of man-hours is also exorbitant.

To address these concerns, we propose a cost model based optimization approach next that does not require multiple executions of the query with different parameters.

5 INSIGHTS INTO DATA ENGINE BEHAVIOR

In the previous section, we saw that the iterative approach is not scalable as it is expensive to run jobs multiple times. Therefore we set upon an alternative approach of using mathematical models to recommend new settings. In this section, we study the effect of each parameter on the query cost by performing a set of experiments. These experiments provided key insights that will be the basis to develop a simple mathematical model. The assumptions made in the iterative approach (Section 4.1) continue to be applicable.

Memory and partition size. We studied the relationship between the container memory and the amount of data processed by the container, i.e. the partition size by performing a set of experiments on the Spark engine. As before, the experiments were run for the TPC DS dataset (scale 1000) and queries on a 4 node AWS cluster with machine type r3.xlarge. We can classify spark tasks into two types based on the data they process. The input mapper tasks process the data from the input tables and the reducer tasks process the output from the previous tasks in the pipeline. The partition size for the input mapper tasks is referred to as the splitsize and is determined by the parameters $split.minsize$ and $split.maxsize$. The partition size for the reducer tasks is determined by the reducer parallelism, which is controlled by the parameter $spark.sql.shuffle.partitions$.

In the first set of experiments, we varied the partition size of the mapper tasks by varying the split parameter while holding all other parameters constant. The splitsize was varied from 150 MB to 600 MB. Based on the container memory, the memory available to each executor core was 1386 MB, which was sufficient to hold even the largest split of 600 MB. Hence, there were no spills in any of the experiments. Figure 4 shows the variation of the cumulative run time with the splitsize. Overall, the cumulative run time reduces as the splitsize increases. This is because the number of mapper tasks reduces as the splitsize increases. This leads to overall lesser cpu overheads in terms of the number of tasks to be managed. Also, reducer tasks down the pipeline have to pull their data from lesser number of mappers, leading to lesser network overheads. Since there are no spills, there is no major negative effect of larger split sizes. It can be further observed that, though the cumulative time reduces, the overall reduction is not large, with the average being 6.73%. This is because the total amount of data processed, IO and network traffic is independent of the parallelism, only the overheads vary.

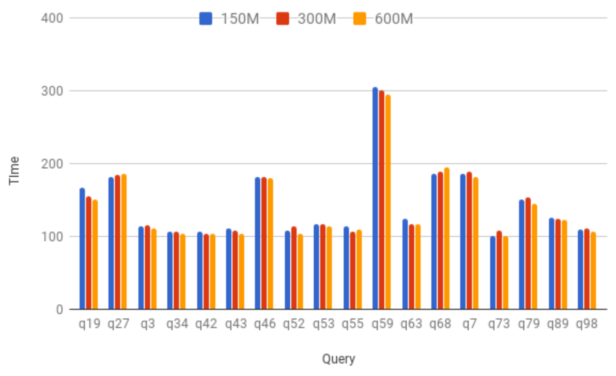

Figure 4: Varying split size [1]

In the second set of experiments, we varied the partition sizes for the reducer tasks by varying the $shuffle.partitions$ parameter from 50 to 800. This varies the number of reducer partitions and correspondingly the amount of data processed by each reducer task. The original TPC DS queries have many filters on the input tables, due to which the mappers process lots of data and the reducers

[1]Bars in chart are in the same order as that of legends

process very less data. The overall time is thus dominated by the mapper tasks. To make this experiment on the reducer parallelism meaningful, we modified the queries to remove all the filters. After this change, the reducer tasks become a significant part of the overall query and changes in reducer times reflect in the overall query times. Figure 5 shows the variations in the cumulative time with the number of partitions. As before, the overall time for the query increases with parallelism, due to increased overheads. The increase is moderate, for e.g., the 16 times increase in parallelism leads to an average slowdown of 47%. These experiments lead to the next insight:

Insight 1. *The parallelism of a query can be increased without significant adverse effect to the cumulative time. At the same time, it should not be increased indiscriminately since the increased parallelism leads to increased overheads and increases the overall cumulative time.*

The increased overheads are due to managing a larger number of tasks and increase in the number of communicating channels. For MR engine, the overhead of starting a JVM for each task can further add to the overhead.

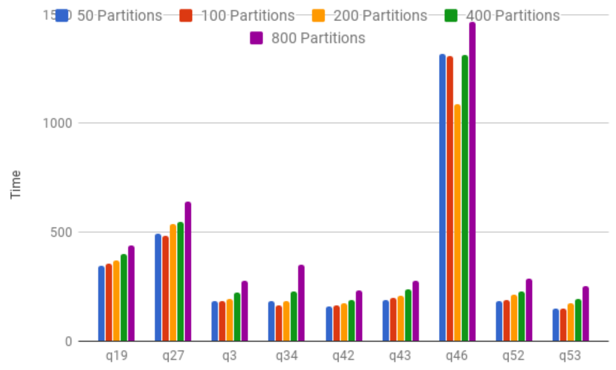

Figure 5: Varying reducer partitions [1]

In some of the experiments with the reducer parallelism, we could observe the effect of spills on the query time. For example, $q27$, $q34$ and $q47$ cumulative time decreased when parallelism increased from 100 to 200 partitions. Figure 6 shows a graph for $q46$, in which the time reduces by almost 30% when the number of shuffle partitions increases from 100 to 200. This is due to the fact that at 100 partitions, the amount of data processed by the reducer task does not fit in memory and causes a spill. As the number of partitions increases to 200, each partition becomes smaller and fits in memory, thus avoiding a spill. This leads to the following insight:

Insight 2. *Spills are expensive as each spill leads to an extra write and read of the data. Thus, spills should be avoided at all costs.*

Spills can be avoided by providing adequate memory to each task or by making more fine grained tasks. Since the memory for each container is fixed, we can either reduce the splitsize (for input mapper tasks) or increase the reducer parallelism (for reducer tasks) so that each task processes only data that can fit in its available memory without causing a spill.

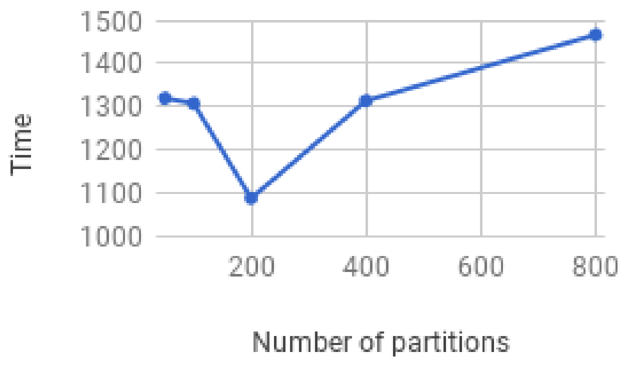

Figure 6: Effect of spill

Memory and instance type. Cloud platforms provide a variety of machines which differ in various characteristics such as cpu speed, disk speed, network speed, cpu cores and memory. Usually, machines with higher memory/cpu core are more expensive for the same CPU type. To evaluate if it worth paying extra for higher memory/cpu core, we conducted a set of experiments where we varied the memory available per container while keeping all other parameters fixed. The experiments were run on a Spark engine on the TPC DS dataset (scale 1000) and using the TPC DS workload. The average data read by each query was around 20 to 40 GB. The experiments were run on a 4 node cluster on AWS with machine type r3.xlarge. We varied the memory per executor core from 746 MB to 1386 MB in steps of 200 MB. The partition sizes were at the default value of 128 MB, so there is enough memory in all cases to hold the data and there were no spills. In most cases, it was observed that the cumulative cpu time usually decreases as the container memory increases. The decrease is due to lesser pressure on memory and reduced costs of garbage collection. However, in some cases the time also increases with memory. In either case, the variation in cumulative cpu cost with memory was not very significant. Figure 7 shows the variation in the cumulative time with memory. Overall the average speedup was 3.5% and the maximum speed up was 18% ($q3$).

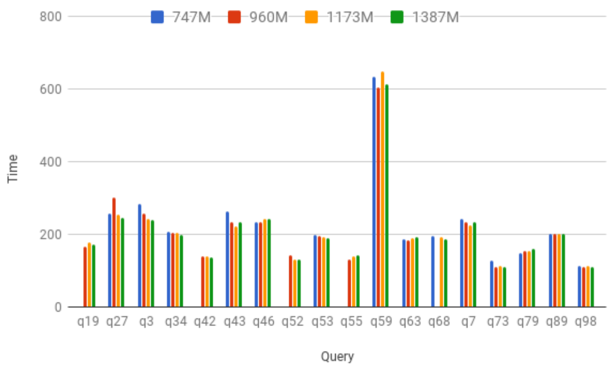

Figure 7: Varying memory per executor core [1]

This behavior is due to the way memory is used by the SQL engines. The memory is mainly used to hold the partition processed by the container (as io.sort.mb in MR and to hold the rdd in Spark). Once there is enough memory to hold the partition in memory (thus avoiding spills), any increase beyond that does not have any additional benefit. As long as tasks do not spill, the total work done in terms of IO, CPU and network traffic is independent of the available memory and the parallelism factor of the tasks. This leads to the following insight:

INSIGHT 3. *For a query, the memory per task can be decreased safely without performance degradation by correspondingly decreasing the partition size (increasing parallelism) if needed to avoid spills.*

This implies that if a job can be tuned to avoid spills on a cheaper instance with same compute but lesser memory than original instance, then it is generally a good idea to move to cheaper instance for saving cost without any performance degradation.

Cores per executor. For Spark engines, there is an additional parameter of cores per executor. Given a certain number of cores per machine, we have a choice of either running many executors with fewer cores per executor (thin executors), or fewer executors with more cores per executor (fat executors). We studied the effect of various choices by varying the $spark.executor.cores$ from 1 to 8 and correspondingly varying $spark.executor.memory$ from 1152MB to 11094MB. The memory per executor core is the same in all cases. Figure 8 shows the variation in the cumulative time with number of cores. It can be observed that fat executors generally provide better performance. On an average, there was a 25% speedup on using 8 cores per executor compared to 1 core per executor. Fatter executors perform better because of several reasons such as better memory utilization across cores in a executor, reduced number of replicas of broadcast tables and lesser overheads due to more tasks running in the same executor process. This leads to the following insight specific to Spark engines:

INSIGHT 4. *Use a single fat executor for each node that uses up all the cores on the node rather many thin executors.*

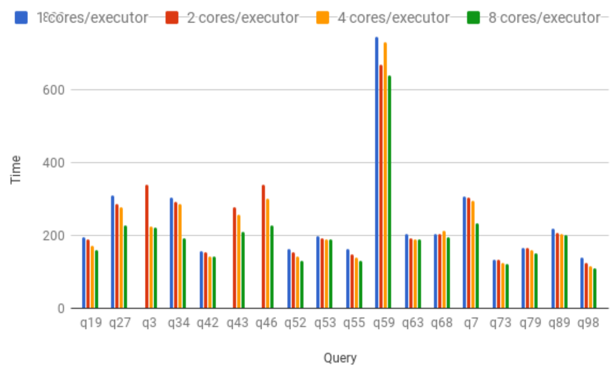

Figure 8: Varying cores per executor [1]

[1]Bars in chart are in the same order as that of legends

Parameters	Description
mapperTime	Total mapper time in seconds
numOfMapper	Number of map tasks
mapperMemory	Container memory for map tasks
splitSize	Input Split Size
mapperInputBytes	Map input in bytes
mapperOutputBytes	Map output in bytes
mapperOutputRecords	Number of Map output records
reducerTime	Total reducer time in seconds
numOfReducer	Number of reduce tasks
bytesPerReducer	Corresponds to Hive parameter *hive.exec.reducers.bytes.per.reducer*
reducerMemory	Container memory for reduce tasks
ioSort	Total amount of buffer memory in mega bytes to be used for sorting. Corresponds to Hadoop parameter *mapreduce.task.io.sort.mb*
spilledMapBytes	Bytes spilled in Map tasks
spilledRedBytes	Bytes spilled in Reduce tasks

Table 2: Job metrics and parameters

6 MODEL BASED APPROACH

We now propose a model for SQL-on-Hadoop engines that builds on the insights described in the previous section. We will describe the model for Apache Hive on Hadoop2 MR and Apache Spark. We will show that even though the model is simple, it is a sufficient approximation to generate good recommendations.

6.1 Algorithm

To model the behavior of a query, we need to know the characteristics of the data processed by the query. This includes the input data size and the output data size for each MR stage of the query processing pipeline. One way to estimate this is to use statistics such as sizes of the tables, number of distinct values for each attribute and histograms that will enable us to estimate selectivities of various operators in the query. However, getting accurate data statistics in a big data environments is very often a challenge. Since we are mainly concerned with ETL queries, we can exploit the fact that these queries are run periodically. SQL-on-Hadoop engines collect a lot of metric and configuration information from the jobs that are executed in the system. The overall approach is thus to use the data collected during a run of the query as inputs to our algorithm to recommend good configuration parameters for future runs of the query. The job metrics and parameters used by the algorithm are listed in Table 2. The algorithm also takes as input information about the machine instance type and configuration, as listed in Table 3. Besides these, there are some global parameters that can be used to tune the algorithm, as listed in Table 4.

Algorithm 2 optimizes cumulative resource utilization of a SQL query. It can be seen from Equation 1 that resource utilization can be reduced by decreasing either the memory usage or time taken for processing. The algorithm thus aims to reduce the memory

Parameters	Description
nodeMemory	Total available memory per node for MR job
cpuPerNode	Number of CPUs per node
vCpuPerNode	Number of vCPUs per node
eCPU	computing units per node similar to Amazon ECU

Table 3: Instance configuration

Parameters	Description	Default
ioSortFrac	Size of ioSort buffer specified as fraction of mapper memory	0.4
maxIOSort	Maximum value for *mapreduce.task.io.sort.mb*	2047
reducerFrac	Fraction of Reducer memory to be used as buffer	0.4
minMapTime	Minimum time that a mapper should take	60s
minRedTime	Minimum time that a reducer should take	60s

Table 4: Global Parameters

usage without increasing the time. Insight 3 indicates that memory usage can reduced without adverse effect by increasing parallelism if necessary. Insight 1 further says that parallelism should not be increased indiscriminately. Thus the algorithm maintains a lower bound on the mapper and reducer time (*minMapTime* and *minRedTime*). It first computes the rate of processing of the mappers (line 1), based on the time metric of the previous run and uses it to compute the splitSize, such that each mapper takes at least *minMapTime* (line 2). It then computes the *ioSort*, such that the output of each mapper fits in that buffer to avoid spills as per Insight 2 (line 3–4). The output size of a mapper is estimated using metrics from the previous run (line 3). The mapper memory is computed from the *ioSort*, using the global parameter *ioSortFrac* (line 5). Similar computations are done to determine the reducer parallelism (*bytesPerReducer*) and *reducerMemory* (lines 6–9). Finally, the expected resource usage based on the new parameters is computed (line 10–12).

6.2 Results

We evaluated the effectiveness of *OptResource* method by running experiments on real workloads. The experiments were carried out for a HIVE on MR engine for a workload consisting of 4 real customer queries (Customer 2). Figure 9 shows the benefit predicted by our model and the actual observed benefit for these queries. The results show that the algorithm leads to significant savings in the cumulative resource usage cost, ranging from 70% for $q3$ to 90% for $q2$. Further, the actual savings closely match the predicted savings indicating that the model is reasonably accurate.

We also evaluated effectiveness of the same technique on SparkSQL. Algorithm for it is very similar to *OptResource* and we are skipping it for brevity. We ran it on 3 real world SparkSQL customer queries (Customer 3). Figure 10 shows that the predicted percent reduction in cost matches that of actual reduction is cost observed with very high accuracy.

Algorithm 2 OptResource

Input: $\mathcal{P}$ is the job metrics and parameters (defined in table 1) from one run, I is instance configuration (defined in Table 3) on which $\mathcal{P}$ is collected, $\mathcal{G}$ is the global parameters defined in Table 4

Output: New job parameters $\mathcal{P}_{new}$ and the expected cumulative resource usage $expectedResUsage$ after optimization.

```
1:  $mapTimePerByte \leftarrow mapperTime/(mapperInputBytes + spilledMapBytes)$
2:  $\mathcal{P}_{new}.splitsize \leftarrow minMapTime/mapTimePerByte$
3:  $mapOutPerSplit \leftarrow \mathcal{P}_{new}.splitsize \times (mapperOutputBytes/mapperInputBytes)$
4:  $\mathcal{P}_{new}.ioSort \leftarrow mapOutPerSplit$
5:  $\mathcal{P}_{new}.mapperMemory \leftarrow \mathcal{P}_{new}.ioSort/ioSortFrac$
6:  $redTimePerByte \leftarrow reducerTime/(mapperOutputBytes + spilledRedBytes)$
7:  $dataPerRed \leftarrow minRedTime/redTimePerByte$
8:  $\mathcal{P}_{new}.bytesPerReducer \leftarrow dataPerRed \times (mapperInputBytes/mapperOutputBytes)$
9:  $\mathcal{P}_{new}.reducerMemory \leftarrow dataPerRed/reducerFrac$
10: $numMappers \leftarrow mapperInputBytes/\mathcal{P}_{new}.splitsize$
11: $numReducers \leftarrow mapperInputBytes/\mathcal{P}_{new}.bytesPerReducer$
12: $expectedResUsage \leftarrow numMappers \times minMapTime \times \mathcal{P}_{new}.mapperMemory + numReducers \times minRedTime \times \mathcal{P}_{new}.reducerMemory$
13:
14: return $\mathcal{P}_{new}$, $expectedResUsage$
```

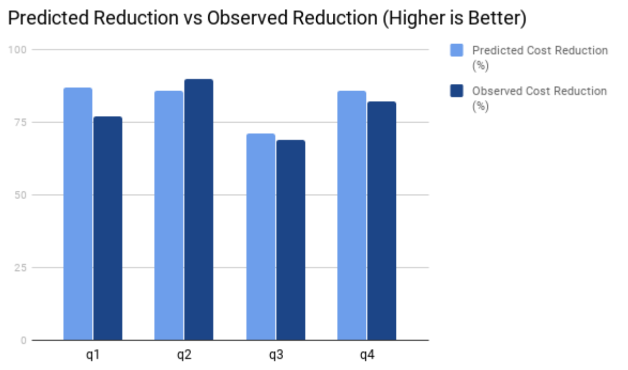

Figure 9: Model Based Result: Predicted reduction in cost versus Actual reduction in cost for HIVE on MR queries

7 MODEL FOR CLOUD PLATFORMS

We will now consider the usage of model based methods for SQL-on-Hadoop engines on cloud platforms. Cloud based platforms have their own peculiarities that necessiates revisiting the algorithm proposed in the previous section.

7.1 Characteristics of Cloud platforms

Cloud platforms provide additional flexibility due to the elastic nature of the cloud and the choice of different instance types.

Cloud elasticity. In the cloud, the physical resource for the cluster are not committed forever. Clusters can be spun up and shut down on demand. In the iterative and model based experiments

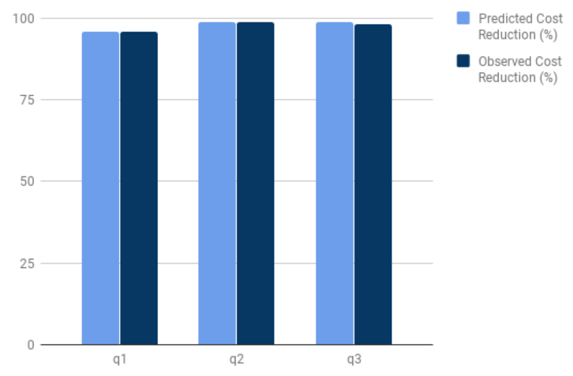

Figure 10: Model Based Result: Predicted reduction in cost versus Actual reduction in cost for SparkSQL queries

till now, we have focused on the cumulative resource usage metric (Equation 1), which was mainly achieved by reducing the memory usage of the containers. In practice, this saving in memory would be useful only if it could be gainfully used by some other task or job. However, for all the big data SQL engines, the memory used by all the task processes is the same. It is controlled by parameters at the query level (e.g. mapper memory, reducer memory, spark executor memory)and cannot be tuned per task. Further, each workload will use its own dynamic cluster in a Cloud setting, which implies that other workloads cannot use the memory saved either.

Thus saving memory may not be very useful and should not be the main criterion. Let us look at memory allocation in further detail. In SQL-on-Hadoop systems, containers are allocated using Yarn, which allocates based on two dimensions - **memory** and **vcpu**. Each container is given 1 vcpu and some memory. If all the vcpus on a node are allocated and memory is still available, the extra memory cannot be used and is wasted. Conversely, if all memory is used up and extra vcpus are available, those extra vcpus are wasted since they cannot be allocated. Figure 11 illustrates the three scenarios:

- Memory/vCPU ratio of container is same as that of instance; hence, neither memory nor CPU is wasted.
- Memory/vCPU ratio of container is smaller than that of instance; hence, memory is wasted as extra memory of instance is still remaining unused after container allocation.
- Memory/vCPU ratio of container is higher than that of instance; hence, CPU is wasted as extra CPU of instance is remaining unused after container allocation.

This implies that memory for each container should be set according of the Memory/vCPU ratio of the machine which leads to the following conclusion:

Insight 5. *For optimal usage of both memory and vcpus, the memory/vcpu ratio of the containers should match the memory/vcpu ratio of the machine type.*

Thus, the memory configuration parameter (mapper memory, reducer memory or spark executor memory) need not be optimized per query and is based on the machine characteristics. Further, since memory per container is fixed, the target metric to be optimized should be the cumulative cpu time (Equation 2) of the query rather than the resource utilization.

Instance types. Cloud platforms provide a choice of different instance types, each of which may have different dollar costs. Thus, it is important to recommend instance types along with configuration parameters. The metric that matters the most in this case is the total dollar cost (Equation 3).

7.2 Algorithm

Algorithm 3 fitJobMR

Input: $\mathcal{M}$ is the job metrics (defined in table 2), I_{old} is instance configuration (defined in Table 3) on which $\mathcal{M}$ is collected, I_{new} is new instance configuration on which $\mathcal{M}$ needs to be optimized upon, $\mathcal{G}$ is the global parameters defined in Table 4

Output: New Job parameter $\mathcal{P}_{new}$ (defined in table 1) that optimize the cumulative time of containers and predicted cumulative time of containers $\mathcal{T}$.

```
1:  newMemPerCore ← $I_{new}$[nodeMemory] /$I_{new}$[vCpuPerNode]
2:  $\mathcal{P}_{new}$[mapreduce.map.memory.mb] ← newMemPerCore
3:  $\mathcal{P}_{new}$[mapreduce.map.reduce.mb] ← newMemPerCore
4:  ioSort ← newMemPerCore × ioSortFrac
5:  if ioSort > $\mathcal{G}$[maxIOSort] then
6:      ioSort ← $\mathcal{G}$[maxIOSort]
7:  end if
8:  $\mathcal{P}_{new}$[mapreduce.task.io.sort.mb] ← ioSort
9:  outPerMap ← $\mathcal{M}$[mapperOutputBytes] / $\mathcal{M}$[numOfMapper]
10: newSplitSize ← $\mathcal{M}$[splitSize] × ioSort / outPerMap
11: $\mathcal{P}_{new}$[splitSize] ← newSplitSize
12: newBPR ← ($\mathcal{M}$[reducerMemory] × $\mathcal{M}$[mapperInputBytes]) / ($\mathcal{M}$[mapperOutputBytes] × $\mathcal{G}$[reducerFrac])
13: $\mathcal{P}_{new}$[hive.exec.reducers.bytes.per.reducer] ← newBPR
14: predictedMapperTime ← $\mathcal{M}$[mapperTime] × ($\mathcal{M}$[mapperInputBytes] / ($\mathcal{M}$[mapperInputBytes] + $\mathcal{M}$[spilledMapBytes])) × ($I_{old}$ [eCPU] / $I_{new}$[eCPU])

15: predictedReducerTime ← $\mathcal{M}$[reducerTime] × ($\mathcal{M}$[mapperOutputBytes] / ($\mathcal{M}$[mapperOutputBytes] + $\mathcal{M}$[spilledRedBytes])) × ($I_{old}$ [eCPU] / $I_{new}$[eCPU])
16: $\mathcal{T}$ ← predictedMapperTime + predictedReducerTime
17:
18: return $\mathcal{P}_{new}$, $\mathcal{T}$
```

The algorithm consists of two parts – **fitJobMR** (Algorithm3) that finds the optimal parameters on a given instance type for MR Job and *optimizeCost* (Algorithm 5) that searches across instance types to determine the best instance type to be used.

Algorithm **fitJobMR** is analogous to *OptResource*, but also takes the instance type I_{new} as input. Instead of choosing the smallest possible *splitSize* like *OptResource*, it starts with fixing *mapperMemory* and *reducerMemory* based on the memory available per core in the instance type being considered (lines 2–3). Based on this, the *ioSort* is computed (lines 4–8). This is followed by computing the *splitSize*, such that the output of that split partition fits in the *ioSort* and does not cause any spill (lines 9–11). Similarly, the *bytesPerReducer* is computed such that the amount of mapper output data processed by each reducer fits in the reducer buffer without causing spills (lines 12–13). The algorithm also computes the estimated cumulative time. Note that this algorithm optimizes a single Map Reduce stage. But typically queries comprises of multiple Map Reduce stages. In which case, we consider only the most dominant stage in terms of cumulative execution time of containers and optimize it. In case, there are no clear dominant stage or multiple dominant stages, we

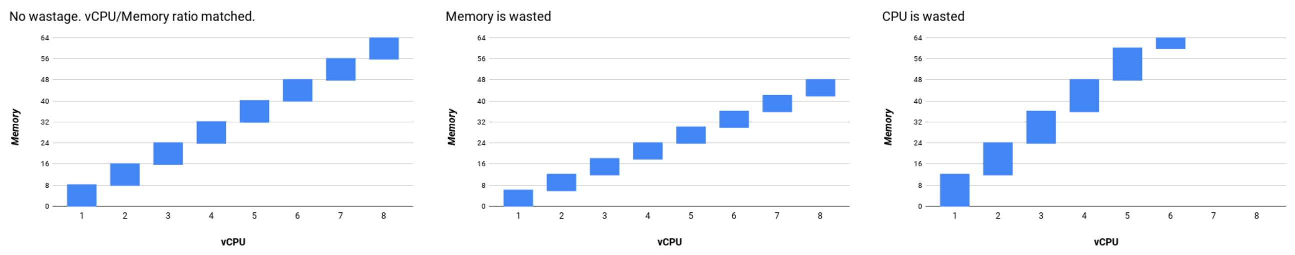

Figure 11: (a) No Resource is wasted (b) Memory is wasted (c) CPU is wasted

are conservative to avoid spills i.e., smallest suggested split size and bytes per reducer across stages are chosen.

Algorithm *optimizeCost* searches for the best instance type. It iterates over all the available instance types and calls **fitJobMR** to find the best configuration and cumulative time for each type (line 2). It computes the dollar cost of the query using the rental rate per core per unit time *r* using Equation 3 (line 3). The instance type leading to the least dollar cost and the configuration corresponding to that instance are finally output by the algorithm.

Algorithm **fitJobSpark** defined in Algorithm 4 is similar to **fitJobMR** but for SparkSQL Jobs instead of Hive on MR jobs. Algorithm **fitJobSpark** takes a set of stage parameters (defined in Table 5) $\mathcal{S}$ as input along with instance and global parameter as in **fitJobMR**.

Parameters	Description
inputBytes	Total Input Bytes Read
shuffleRead	Shuffle Bytes Read

Table 5: Stage Parameter for SparkSQL Job

Algorithm 4 fitJobSpark

```
Input: $\mathcal{S}$ is set of stage parameters (defined in table 5 ), $\mathcal{I}$ is new instance configuration on which Spark Job needs to be optimized upon, $\mathcal{G}$ is the global parameters defined in Table 4, $\mathcal{T}$ is cumulative task time to execute job.
Output: New Job parameter $\mathcal{P}_{new}$ (defined in table 1) that optimize the cumulative time of containers and predicted cumulative time of containers $\mathcal{T}_{new}$.
1: $\mathcal{P}_{new}$[spark.executor.cores] $\leftarrow$ $\mathcal{I}$ [vCpuPerNode]
2: $\mathcal{P}_{new}$[spark.executor.memory] $\leftarrow$ $\mathcal{I}$[nodeMemory] $\times$ $\mathcal{G}$[nodeMemoryFrac]
3: newMemPerCore $\leftarrow$ ($\mathcal{I}$[nodeMemory] $\times$ $\mathcal{G}$[nodeMemoryFrac]) / $\mathcal{I}$[vCpuPerNode]
4: shuffleBuffer $\leftarrow$ newMemPerCore $\times$ $\mathcal{G}$[shuffleBufferFrac]
5: readBuffer $\leftarrow$ newMemPerCore $\times$ $\mathcal{G}$[readBufferFrac]
6: for each $s_i \in \mathcal{S}$ do
7:     splitSize $\leftarrow$ $min$( $s_i$[inputBytes] / readBuffer, splitSize)
8:     shufflePartiton $\leftarrow$ $max$(shuffleRead / shuffleBuffer, shufflePartition)
9: end for
10: $\mathcal{P}_{new}$[mapreduce.input.fileinputformat.split.maxsize] $\leftarrow$ splitSize
11: $\mathcal{P}_{new}$[mapreduce.input.fileinputformat.split.minsize] $\leftarrow$ splitSize
12: $\mathcal{P}_{new}$[spark.sql.shuffle.partition] $\leftarrow$ shufflePartition
13: $\mathcal{T}_{new} \leftarrow \mathcal{T} \times (I_{old}$ [eCPU] / $I_{new}$[eCPU])
14: return $\mathcal{P}_{new}$, $\mathcal{T}_{new}$
```

7.3 Results

We evaluated the new cost metric defined for cloud on a synthetic workload as well as the same SparkSQL customer queries which were evaluated in Figure 10.

Algorithm 5 optimizeCost

```
Input: $\mathcal{P}$ is the job parameters (defined in 1 ), Set $\mathcal{I} = i_1, i_2, i_3, \ldots i_n$ set of instance configurations (defined in 3), rental rates per core per unit time for each type $\mathcal{R} = r_1, r_2, \ldots r_n$
Output: Returns optimized instance type and job parameters across all the instance configurations in $\mathcal{I}$.
1: for $x = 1, 2 \ldots$n do
2:     $config, time \leftarrow fitJobForMR(\mathcal{P}, i_x)$
3:     $cost \leftarrow bestTime \times r_x$
4:     if $cost < bestCost$ then
5:         $bestCost \leftarrow cost; bestInstance \leftarrow i_x; bestConfig \leftarrow config$
6:     end if
7: end for
8: return $bestInstance, bestConfig$
```

Synthetic Workload. For the synthetic workload, we used the TPC DS dataset (scale 1000) and a subset of the queries. The experiments were run on a 4-node cluster on AWS with machine type r3.4xlarge running SparkSQL on an Apache Spark cluster. We chose this specific machine type as it is the most popular machine type in Qubole Data Service. The cloud model suggested that it is cheaper to run these queries on c3 family of machines. We reran the queries on a 4-node cluster with machine type c3.4xlarge. Figure 12 shows the reduction percent on absolute dollar cost of running a subset of the TPC DS queries. Dollar cost is determined by the cost of running a particular AWS instance for cumulative time of running query as defined in Section 3.

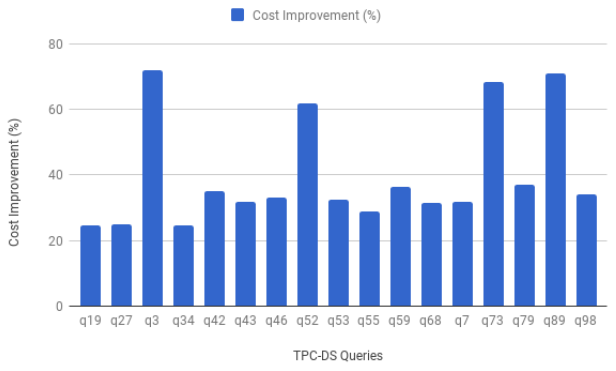

Figure 12: Synthetic workload reduction in absolute dollar cost

Real Workloads. We ran our experiments on Amazon Web Services. Figure 13 shows the reduction percent on absolute dollar cost of running 3 customer queries.

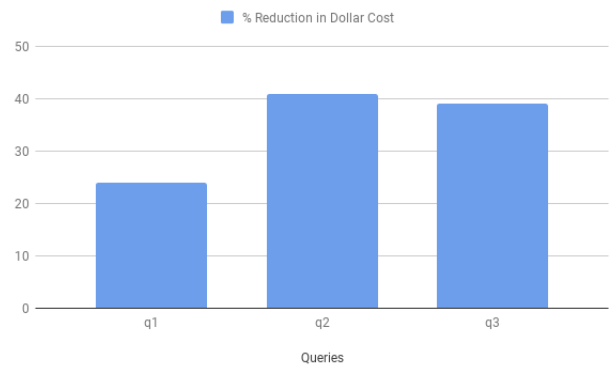

Figure 13: Real workload reduction in absolute dollar cost

8 CONCLUSIONS

In this paper, we have addressed the problem of determining good configuration parameters for SQL workloads on big data platforms, specifically SQL-On-Hadoop engines like Hive on MR and Spark SQL. We started with an iterative algorithm that executes each query multiple times with different configuration parameters. Though this method was able to discover good configurations within a small number (around 10-15) of iterations, we found that it was not feasible for real customers due to the large dollar cost of executing each query multiple times. We then focused on a model based approach and developed models for Hive and Spark SQL based on some insights gained through running many experiments. We further extended the model for cloud platforms that require recommending instance types for the cluster in addition to the other parameters. Our results show that the models were able to provide very good recommendations compared to the default and expert chosen configurations on real customer workloads.

It may sound surprising that very simple models proved to be very effective in practice. This was possible since we chose to focus on SQL-On-Hadoop workloads rather than generic MR or Spark workloads. This enabled us to reason about the query execution and develop the insights. Future work includes extending the models for other engines like Tez and Presto, exploring more configuration parameters and applying the models on many more real workloads by integrating it with the Qubole product.

REFERENCES

[1] Surajit Chaudhuri and Vivek Narasayya. 2007. Self-tuning Database Systems: A Decade of Progress. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*. VLDB Endowment, 3–14. http://dl.acm.org/citation.cfm?id=1325851.1325856

[2] Biplob K. Debnath, David J. Lilja, and Mohamed F. Mokbel. 2008. SARD: A statistical approach for ranking database tuning parameters. In *Proceedings of the 24th International Conference on Data Engineering Workshops, ICDE 2008, April 7-12, 2008, Cancún, México*. 11–18. https://doi.org/10.1109/ICDEW.2008.4498279

[3] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning Database Configuration Parameters with iTuned. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 1246–1257. https://doi.org/10.14778/1687627.1687767

[4] Herodotos Herodotou and Shivnath Babu. 2011. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *Proceedings of the VLDB Endowment* 4, 11 (2011), 1111–1122.

[5] Herodotos Herodotou, Fei Dong, and Shivnath Babu. 2011. No One (Cluster) Size Fits All: Automatic Cluster Sizing for Data-intensive Analytics. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing (SOCC '11)*. ACM, New York, NY, USA, Article 18, 14 pages. https://doi.org/10.1145/2038916.2038934

[6] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. 2011. Starfish: A Self-tuning System for Big Data Analytics.. In *Cidr*. 261–272.

[7] Facebook Inc. 2017. Distributed SQL Query Engine for Big Data. (2017). https://prestodb.io/

[8] Qubole Inc. 2017. Cloud Data Platform for Insights Driven Enterprises. (2017). http://www.qubole.com/

[9] Sandeep Kumar, Sindhu Padakandla, Chandrashekar Lakshminarayanan, Priyank Parihar, K. Gopinath, and Shalabh Bhatnagar. 2016. Performance Tuning of Hadoop MapReduce: A Noisy Gradient Approach. *CoRR* abs/1611.10052 (2016). http://arxiv.org/abs/1611.10052

[10] Palden Lama and Xiaobo Zhou. 2012. Aroma: Automated resource allocation and configuration of mapreduce environment in the cloud. In *Proceedings of the 9th international conference on Autonomic computing*. ACM, 63–72.

[11] Min Li, Liangzhao Zeng, Shicong Meng, Jian Tan, Li Zhang, Ali R. Butt, and Nicholas Fuller. 2014. MRONLINE: MapReduce Online Performance Tuning. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing (HPDC '14)*. ACM, New York, NY, USA, 165–176. https://doi.org/10.1145/2600212.2600229

[12] Raghunath Nambiar. 2014. Benchmarking big data systems: introducing TPC express benchmark HS. In *Workshop on Big Data Benchmarks*. Springer, 24–28.

[13] Meikel Poess, Bryan Smith, Lubor Kollar, and Paul Larson. 2002. Tpc-ds, taking decision support benchmarking to the next level. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. ACM, 582–587.

[14] Adam J. Storm, Christian Garcia-Arellano, Sam S. Lightstone, Yixin Diao, and M. Surendra. 2006. Adaptive Self-tuning Memory in DB2. In *Proceedings of the 32Nd International Conference on Very Large Data Bases (VLDB '06)*. VLDB Endowment, 1081–1092. http://dl.acm.org/citation.cfm?id=1182635.1164220

[15] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1626–1629.

[16] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. ACM, New York, NY, USA, 1009–1024. https://doi.org/10.1145/3035918.3064029

[17] Dili Wu and Aniruddha Gokhale. 2013. A self-tuning system based on application Profiling and Performance Analysis for optimizing Hadoop MapReduce cluster configuration. In *High Performance Computing (HiPC), 2013 20th International Conference on*. IEEE, 89–98.

[18] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. 2016. Apache Spark: A unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.