Reverse Engineering Design Decisions of Query Execution Engines

CSE 544/550: Project Report

Helga Gudmundsdottir, Parmita Mehta
Computer Science and Engineering Department, University of Washington
Seattle, Washington, USA
{helgag, parmita}@cs.washington.edu

Abstract

The landscape of data analytic engines is very diverse with hundreds of open source and commercial options available. These systems vary in complexity and their performance often varies greatly for different kinds of workloads. Some engines are more efficient at executing certain types of queries or perform better at scale. Understanding the execution behavior of these engines, their various underlying design decisions and their implications of performance, requires detailed experimentation and deep expertise in each system. In this paper, we explore whether low-level system metrics, such as CPU load, disk I/O or network utilization, can provide insights into how certain design decisions appear during query execution. Example design decisions that we want to capture are degrees of parallelism; synchronization barriers; whether intermediate results are materialized or data is shuffled. We hypothesize that system-level manifestations of these design decisions is consistent across various query execution engines. Since system level monitoring can be done without special instrumentation, we believe that there is an opportunity for providing rich information about system behavior that is useful for reasoning about performance and efficiency in a variety of scenarios.

In this work, we focus on the query execution layer of Myria, a parallel shared nothing Big Data management system from the University of Washington. We collect measurements while running a synthetic workload consisting of simple building block queries. We analyze the traces to examine disparity in system behavior for different query classes. Using standard machine learning techniques, we build black-box models of these queries and evaluate whether we can automatically classify or cluster types of queries, given only measurement traces.

1. INTRODUCTION

In recent years, there has been an explosion of innovation in data management systems, especially in the context of data analytics and Big Data. There is an incredibly diverse landscape of systems of varying complexity, each with different performance properties and tradeoffs for a huge variety of applications. In practice, evaluation and comparison of systems is done through standard benchmarking or by running a particular workload and comparing overall performance and manageability. However, running benchmarks does not explain why one engine is better at certain tasks than others. What subtle differences make one system better suited for a particular workload? How do we uncover what a system is doing as it executes a query? Today, this usually requires deep expertise in each system, detailed experimentation, knowledge of the code base, instrumentation and profiling. This can be expensive, require access to experts, training of employees, and access to system internals, which may not always be possible. While some systems come with monitoring tools, these tools are of varying quality, are not standardized and rarely available for newer engines.

Given a system executing two different data analytics task, is there a way to automatically infer what the underlying system is doing to accomplish these tasks? We envision a tool that can monitor any query engine and provide rich information about the execution of tasks, without access to system internals. The key hypothesis of this paper is that low-level system metrics, such as CPU load or disk I/O, can provide insight into various design decisions that appear during query execution. Further, we suspect that system-level manifestations of these design decisions is consistent across a variety of different query execution engines. In this work, we explore how we can extract information from low-level measurements, and whether it is possible to automatically learn system-level signatures of different design decisions.

There are multiple design decisions that can make or break the performance of a query execution engine. Perhaps a task was slow because intermediate data was written to disk, or fast because it remained in memory. There could have been CPU intensive operations, or a lot of data was shuffled. Maybe the system suffered from inefficient memory management or there were synchronization barriers. In a distributed setting, maybe the system wasn't able to use the entire cluster, didn't deal well with stragglers, or some nodes behaved abnormally, perhaps due to any number of failures that might arise. When we understand what these systems are really doing, we can start to reason about what implications these decisions have on performance and efficiency. Such insight is at the heart of many important tasks:

- **System debugging and optimization**: Information about design decisions can be used by developers or administrators to focus debugging efforts; identify performance bottlenecks; evaluate differences between system versions or resource configurations; or automatically detect abnormal behavior and failures.

- **Query tuning**: Understanding what the underlying system is doing when executing a query can help users make appropriate changes to the analytics task or system configuration.

- **System comparison**: With the confounding list of options available today, being able to gain rich information at the level of design decisions without special instrumentation can greatly reduce the effort needed in evaluating which system is best suited for particular tasks. Such fine grained information can even enable administrators to split workloads among many systems, or serve as input for adaptive scheduling in a polystore environment.

In this paper, we explore the feasibility of a tool that is able to automatically capture and expose these design decisions to users, providing rich information about the operations of the system in question using only low-level system measurements. We focus on the query execution layer of Myria[3], a parallel shared nothing Big Data management system from the University of Washington. We analyze measurements collected while running a synthetic workload consisting of simple building block queries. The queries are divided into three broad classes (*select*, *join* and *aggregate*) and have different filter operations on rows and columns. We evaluate various machine learning techniques and their effectiveness at distinguishing between different query class, *i.e.* whether a set of measurements correspond to a particular type of query class. We build black-box models for classification and clustering and achieve 70-80% overall accuracy.

2. RELATED WORK

The use of benchmarking, log analysis, profiling and tracing can give valuable information about the behavior of systems. Profiling is achieved through instrumentation of either source code or program binaries. An example of a tracing framework is H-trace[1], which is based on Google's Dapper system[9]. It is intended to be used to monitor large-scale distributed systems in production and also requires instrumentation of code. Analyzing logs can provide insights, but their format is not standardized across systems, and low-level system metrics usually have to be collected separately and later correlated with log entries. Our approach is fundamentally different from profiling and tracing, as we aim to distill low-level system metrics into higher-level design decisions. Our goal is not to replace all other tools, but provide a new and general way of analyzing systems, that can be complementary to other methods, for example by focusing debugging efforts or automatically detecting abnormal behavior.

To the best of our knowledge, there is no prior work that tries to build black box models from system metrics that aim to predict or reason about the behavior of DBMS's. The most closely related work we have found centers around building models for performance prediction. Buffer access latency was determined to be an accurate predictor of query performance for concurrent OLAP workloads[1]. Both white box and black box models have been used to predict the performance of OLTP workloads[6, 7]. The white box model is built for MySQL and utilizes deep knowledge of design and implementation decisions while the black box model uses system level measurements. They found the black box models to be quite effective in predicting resource utilization for resources like CPU, network and log writes for concurrent OLTP queries.

OS-level metrics have also been used to detect and analyze performance problems and faults in a variety of systems. Pan et. al use such black-box diagnostics to detect performance problems in Hadoop[8]. Their approach is to look for asymmetric behavior across nodes, with the assumption that nodes behave similarly in the absence of performance problems and an outlier is thus a likely cause for problems.

[1] `http://htrace.incubator.apache.org/`

3. WORKLOAD

Dataset. For our experiments we used the TPC-H star schema benchmark[2] to generate datasets at multiple scale factors. The TPC Benchmark (TPC-H) is a decision support benchmark, which consists of a suite of business oriented ad-hoc queries and concurrent data modifications. The queries and the data populating the database have been chosen to have broad industry-wide relevance and is widely used in research. The generated dataset consists of one large fact table and four smaller dimension tables. We initially fixed the size of the data set to 2GB. However, the query runtimes were overall very short, with some queries completing within a single second. This limited our analysis and the resulting classification accuracy was poor. We therefore increased the dataset to 10GB, giving us longer running queries and a more uniform distribution of query runtimes. The results discussed in this paper are all based on this larger dataset, unless otherwise stated.

Queries. We used TPC-H to generate a dataset, but decided to construct our own queries as the TPC-H queries are too complex for our learning goals. Our constructed queries fall into three broad classes – *select*, *join*, and *aggregate* – and have different filter operations on rows and columns. We designed the queries to be very simple in order to limit the number of underlying operators used during query execution. This allows us to better isolate fine grained behavior, simplifies the operation-to-query mapping and overlap of the underlying operations for each of the queries. For example, all queries utilize the `DbQueryScan` operator, and a simple *select* query isolates the behavior of that operator. A *join* query, on the other hand, uses the `SymmetricHashJoin` operator in addition to `DbQueryScan`, and then shuffles results. Table 1 provides an overview of the different queries we ran.

Data partitioning. For all of the *select* and *aggregate* queries, the data is horizontally partitioned across workers. For *join* queries we utilize the dimension tables replicated on each of the workers instead of utilizing the horizontally partitioned dimension tables. This was done to mimic the recommended production configuration of large datawarehousing environments.

4. EXPERIMENTS

We ran our experiments on Amazon Web Services EC2 with a fixed configuration for Myria. We deployed several three node clusters, with one master node and two worker nodes. Each node has the same amount of resources, is an `m1.large` instance type with 2 CPUs and 7.5 GB memory.

System-level monitoring. During experiments, we collect low-level measurements at worker nodes. Initially, we used Ganglia[3], a widely used framework for monitoring distributed systems, for this task. However, while Ganglia is well suited for monitoring distributed systems in production over long periods of time, we encountered some critical limitations for using it for our project, as the data resolution was not fine grained enough for short term experiments. We therefore decided to use Sysstat[4] for metric collection, a tool which runs on any Linux based machine and functions as a front end to the kernel's `/proc` filesystem. Sysstat gives us single second data resolution and a huge number of different metrics to monitor, but does not does not come with any distributed

[2] `http://www.tpc.org/tpc_documents_current_versions/pdf/tpch2.17.1.pdf`

[3] `http://ganglia.sourceforge.net/`

[4] `http://sebastien.godard.pagesperso-orange.fr/`

Class	Name	Description
Aggregate	agg_count_1	Number of tuples in fact table
	agg_avg_1	Average orderkey from fact table
	agg_count_1..4	Number of tuples from dimension tables 2, 3 and 4
	agg_avg_1..4	Average value for key column for dimension tables 2, 3 and 4
Join	join_factdim_1..4	Join fact table with each of the dimension tables
	join_multi_3..5	Join dimension table 2 and 4 with the fact table
	join_range_1..2	Join fact table with dimension tables 1 and 3, and filter rows on fact table orderkey
Select	select_all_1..5	Select * from fact and each of the dimension tables
	select_1..80_all	Select orderkey from fact table filtering 1,10,20,40,80 % of the rows
	select_project_1..2	Select all rows and some columns from fact table
	select_range_1..2	Select all rows and one column from fact table
	select_rangeproject_1..2	Select rows and columns from the fact table

Table 1: Queries by query class

measurement collection framework, which was an attractive feature of Ganglia. We therefore implemented a simple framework for monitoring a cluster remotely, which deploys Sysstat locally at worker nodes collects results at certain time intervals so as not to interfere with running experiments.

Data collection. The primary goal of the experiments is collecting measurement data, where each "record" corresponds to a single execution of a particular query. We want these executions to be run in as much isolation as possible, so the measurements are not affected by other executions of queries. We therefore wait 90 seconds before issuing a query to make sure that any metrics that are calculated as an average over time, such as average system load, are not affected by previous executions. We also flush both OS and database caches between each execution. While we ensure that the Myria worker process is the primary source of activity on each node during experiments, we do not try to control for incidental variance due to external factors, such as Java garbage collection, multi-tenancy, hypervisor implementations, EC2 cluster configuration, etc.

For the final 10GB dataset and approximately 30 different queries presented in this paper, we collected around 2,500 labeled query executions. For each of these executions we monitor over 60 metrics at workers and store the resulting measurements in a database along with information about query execution details. We believe this gives us a sufficiently large sample for the purposes of this initial exploratory study.

5. ANALYSIS

Each of the experiments provided sequence data for sixty-one metrics on two workers. To simplify analysis we extracted features from this sequence data to capture magnitude, range and distribution of the metric over the duration of a query execution.

For each metric on a particular worker during a single execution, we extract summary statistics: minimum, maximum, median, mean, standard deviation and sum, which represents the area under the curve (since measurements were taken at one second intervals). This gives us a total of three hundred and sixty-six features per worker for each query execution. Note the distinction between a metric (*e.g.* CPU idle time) and a feature (*e.g.* the max value of CPU idle time). We considered various ways of combining the worker specific features into a single feature vector, such as combining and calculating ratios, or even treating each worker as a separate record of an execution. For the purposes of this initial study, we decided to average the features across the two workers, and added query execution duration as a feature. This resulted in a feature vector of three hundred and sixty-seven elements.

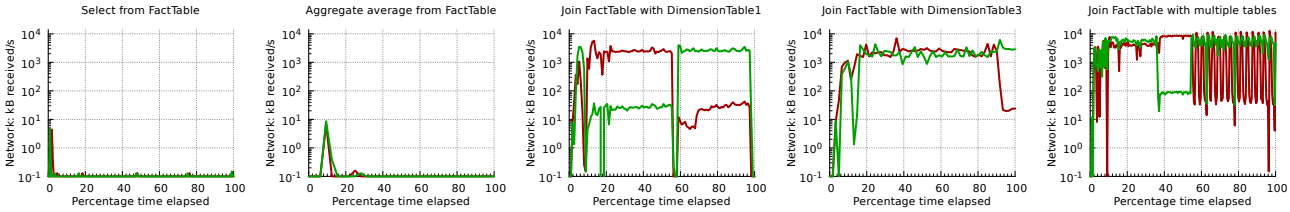
Further reducing the dimensionality of our data is an important step. Multiple dimensions are hard to think in and impossible to visualize. Due to the exponential growth of the number of possible values with each dimension, complete enumeration of all subspaces becomes intractable with increasing dimensionality[5]. With a feature vector of 367 attributes, we anticipate difficulty in detecting signatures for query classes. The concept of distance becomes less precise as the number of dimensions grows, since the distance between any two points in a given dataset converges. Therefore, our next step was to reduce the number of features in the feature vector without losing information from the dataset. An optimal feature set would be very important in providing insights into design decisions, therefore we decided to try two different approaches for selecting features. As the first approach we used statistical distribution and correlation as well as systematic analysis to manually pick the most relevant features. For the second approach we utilized machine learning algorithms to extract relevant features. Both of the approaches are discussed in the following section.

5.1 Measurement Analysis

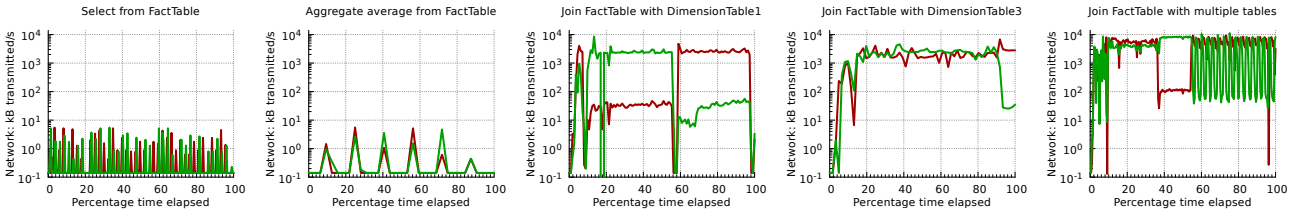
After having extracted features from sequences of measurement data as described above, we systematically examined the measurements. Although we also use automatic methods of feature selection, we want to compare such results with human-in-the-loop findings. We want to better understand which metrics are important and which are not, and use that information to further examine system behavior, remove highly correlated metrics and perhaps reduce monitoring overhead in later experiments. In trying to infer what information about system behavior and manifestations of design decisions in these metrics, we also looked at the sequenced data as timeseries graphs for several samples of execution runs. Some of the more interesting findings are shown in Figure 1.

For each of the features in our dataset, we scale all values to a range between zero and one to standardize the range of different metrics. We then remove a total of 135 features that have zero variance. In many cases, features that correspond to the minimum value of a metric that records intermittent activity, such as disk transfers per second, have zero variance. A total of 19 metrics showed no variance for any of their extracted features. Most notably, we observed that no swapping occurred at any point during our experiments and HugePages were never used, an unsurprising fact given our familiarity with the Myria implementation.

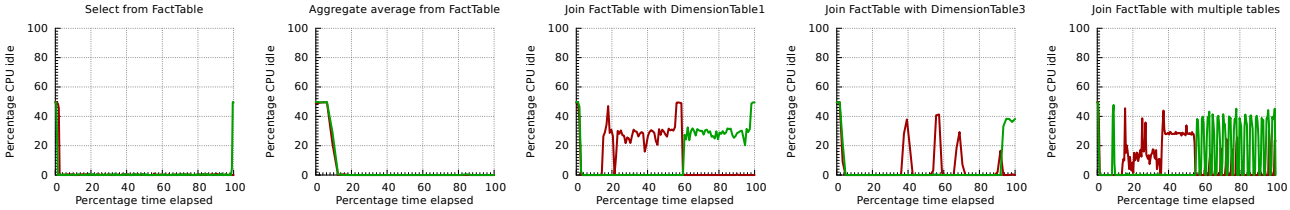
With the remaining 232 features we calculated chi-squared test scores. The chi-squared test measures dependence between stochastic variables, and is helpful to identify variables that are



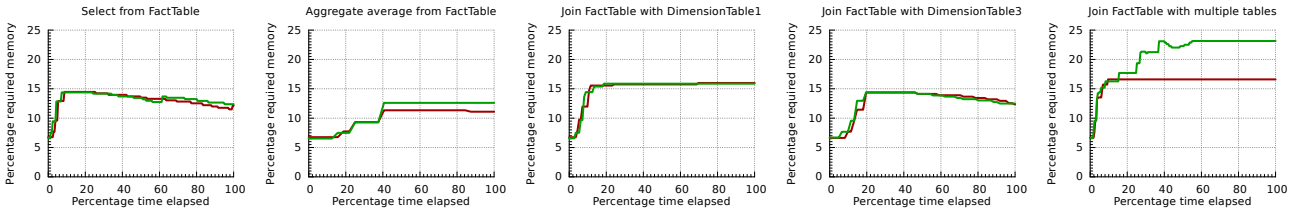
(a) **Received traffic on non-loopback network devices** (notice the logscale on the y-axis). The *select* and *aggregate* queries behave similarly, receiving the query at the start of the series, but then nothing more. The difference in behavior of the two different *join* queries was surprising. In one of them, there is a clear separation of roles between the workers, as one receives large amount of data from the other up until midpoint, where they switch role. The expected behavior was as in the second query, where the communication is mostly equal throughout. There is also a three part segmentation in the query with multiple *joins*.



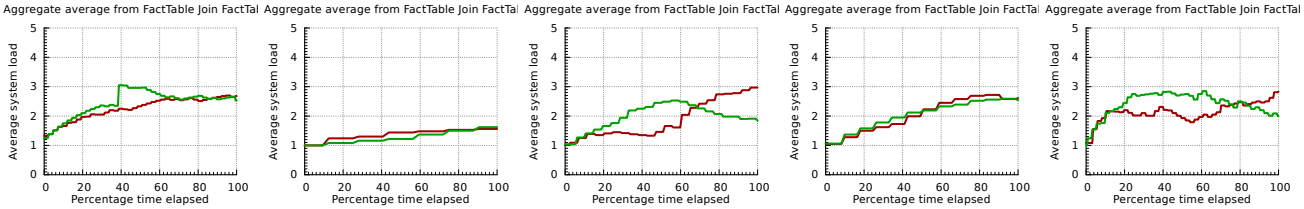
(b) **Transmitted traffic on non-loopback network devices** (notice the logscale on the y-axis). Here we see the *select* and *aggregate* queries sending small amounts of data intermittently, most of which are likely heartbeat messages to the master node. For the *join* queries we see the same behavior as described in (a), confirming that the workers are communicating with one another.



(c) **Percentage of CPU idle time**. In the *select* query we see that the green worker starts executing the query slightly faster than the red worker, and completes slightly sooner as seen by the spike at the end. In this metric, we are better able to see the division of labor between workers in the first *join* query. For the for half, the green worker has full CPU utilization, while the red worker spends a lot of time idling. At around the 60% mark, their roles reverse. This poor balance can also be noticed in the multi table *join*, but is much better in the *join* with DimensionTable 3.



(d) **Amount of memory needed for the current workload**, as a percentage of total memory, is very similar across all of the queries. For the multi-*join*, there is a noticeable imbalance between the worker, with the green worker requiring more. It seems that the red worker is able to load everything into memory quite early, but the memory needs of the green worker keep increasing until half-way through the query, corresponding to the time where CPU idle time starts rising.



(e) **System load average for the last minute**, calculated as the average number of runnable or running tasks, and the number of tasks in uninterruptible sleep. There is a noticeable spike in system load for the green worker in the *select* query. This is unusual and likely due to external factors. Notice that a corresponding spike does not appear in the other metrics that we have examined for this query execution. With this metric we are very clearly able to see the balance, or imbalance, of work between workers.

Figure 1: **Sequenced measurement data of executions of five selected queries.** For each of the queries, we show several metrics that all correspond to the same execution of that query. These query executions were selected as follows. Out of all executions for a particular query, we selected five samples at random. We then looked at the timeseries of each of these samples for several different metrics and chose executions that are representative for each query. We show here a few of the more interesting queries and metrics observed. The green and red lines correspond to measurements taken at the two workers. Their colors are consistent for all metrics corresponding to a particular query.

likely to be independent of class and therefore irrelevant for classification. We calculated these scores for a few different class labels, the primary being the three broad query classes (Select, Joins, and Aggregates), but also for subclasses (*e.g.*. Join followed by a project), and binary classifications (*e.g.* whether the query was an *aggregate* or not). We use these scores to guide our process of manually going through metrics.

Network. Features with the highest chi-squared score for predicting the query class or subclass are all network related. The top two metrics correspond to the amount of traffic received or sent over the network per second. All features derived from these metrics rank highly, except the minimum, which is likely zero in most cases. We decide to discard all other network metrics as we believe that the transfer rates provide a sufficiently detailed view of network activity. Figures 1a and 1b examples of how these metrics appear during query execution.

CPU. The most prominent CPU related metric is idle time, which the percentage of time that the CPU was idle and the system did not have an outstanding disk I/O request. While this metric ranks highly for predicting both query classes and subclasses, it ranks highest for a binary classification of whether a query was an *aggregate* or not. Other CPU metrics do not rank as highly, although the percentage of time that CPU was idle while waiting for I/O was chosen by automated feature selection algorithms as described below. One would expect the percentage of CPU utilization that occurred while executing at the user level to be descriptive of how computationally heavy a query is. Given the simplicity of our queries, we probably don't capture this aspect.

Memory. Metrics directly related to memory utilization don't rank very highly. A notable metric is the amount of memory needed for the current workload. This is an estimate of how much RAM/swap is needed to guarantee that it never runs out of memory, and is therefore an interesting metric to capture how memory hungry queries are. We also include the amount of active memory, which is memory that has been used recently and usually not reclaimed unless absolutely necessary, but discard several other related metrics, such as amount of memory used, free and inactive. The amount of memory used as buffers is selected by automatic feature selection, but does not rank highly here. The amount of memory used by the kernel to cache data, which we clear before the start of each execution, also ranks fairly low. This metric may be more interesting to a workload that consists of more complicated queries.

Disk I/O. We observe that metrics related to disk traffic are largely irrelevant to our classification tasks. This is not surprising, given that most of our queries will read a similar amount of data into memory during a scan operation (we flush database caches before the start of each query). With 7.5GB RAM at each worker and a 10GB total size of data, there is no memory pressure.

System. There are various other metrics that rank highly, such as the number of interrupts per second and the number of page faults made by the system per second. This is not a count of page faults that generate I/O, because some page faults can be resolved without it. Average system load is also interesting, especially for longer running queries. The load average is calculated as the average number of runnable or running tasks, and the number of tasks in uninterruptible sleep over a one minute interval. Although not directly captured by our method of extracting summary features from measurements, this load average gives insight into the load distribution across workers.

Attribute	Description
`network_txpck_s_mean`	Transferred packets (mean)
`cpu_iowait_sum`	CPU I/O wait (sum)
`paging_vmeff_sum`	Virtual memory efficieny (sum)
`intr_intr_s_median`	Interrupts per Second (median)
`network_rxpck_s_median`	Received Packets (median)
`memutil_kbbuffers_max`	Memory as buffers (max)
`duration`	Execution time of the query

Table 2: Features selected by Weka

5.2 Feature Selection

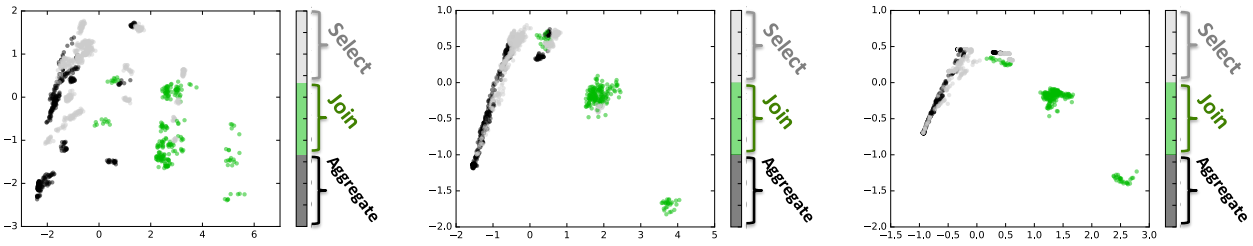
We used Weka[2] for machine learning based feature selection. The process for feature selection in weka is separated into two parts, *attribute evaluation* used to assess attribute subsets, and a *search method* used for searching the space of possible subsets. We tried a variety of the evaluator and search methods to select features and used these features to build and test a classification model. We found `CfsSubsetEval` and `RankSearch` with `GainRatioAttribute` evaluator to provide the best set of features for classification accuracy. `CfsSubsetEval` evaluates the worth of a subset of attributes by considering the individual predictive ability of each feature along with the degree of redundancy between them. Subsets of features that are highly correlated with the class, while having low inter-correlation, are preferred. For search we used `RankSearch`, which ranks all attribute using `GainRatioAttribute` evaluator. `GainRatioEvaluator` evaluates the worth of an attribute by measuring the gain ratio with respect to the class label. From the ranked list of attributes, subsets of increasing size are evaluated. This approach gave us seven attributes as listed in Table 2.

PCA. Principal Component Analysis (PCA) is a well known technique for finding correlations in a single multi-variate dataset. PCA identifies dimensions of maximal variance in a dataset and projects the raw data onto these dimensions. We used PCA as a tool to visualize the effectiveness of our different approaches to feature selection. We took the features selected by each of the approaches and used PCA to reduce it to two components. Figure 2 shows these PCA derived two dimensions overlaid with query class labels and provides a rough visualization of query class clusters. For the baseline, in Figure 2a we use all attributes that showed non-zero variance. This method does poorly in identifying the three clusters, although *joins* are generally grouped together. We then use handpicked metrics from the measurement analysis (Figure 2b) and features picked by machine learning (Figure 2c). We observe that both of these approaches of feature selection consolidated the query class clusters. However, neither approach was able to separate the clusters for *aggregate* and *select* queries, which overlap significantly. Both of these methods also identify two main clusters of *join* queries.

6. EVALUATION

6.1 Prediction

To evaluate our approach, we build machine learning models from the selected features and evaluated their accuracy. We used Weka[2] for automated feature selection and building machine learning models. Weka is a popular suite of machine learning software written in Java, developed at the University of Waikato, New Zealand. We build a multi-class classification model as well as an unsupervised clustering model as described below. The per query



(a) All features. (b) Manually selected metrics. (c) Weka selected features.

Figure 2: PCA based visualization of the effectiveness of different approaches to feature selection for predicting the three query classes. All methods do poorly at distinguishing between *select* and *aggregate* queries, while *joins* are mostly grouped separately.

Query Class	Select	Aggregate	Join	Weighted Avg.
TP Rate	0.803	0.644	0.891	0.78
FP Rate	0.234	0.107	0.017	0.138
Precision	0.727	0.715	0.95	0.783
Recall	0.803	0.644	0.891	0.78
AUC	0.843	0.865	0.986	0.888

Table 3: Detailed accuracy by query class for classification model

Q.Class/Cluster	Cluster 0	Cluster 1	Cluster 2
Select	85.06%	2.04%	12.90%
Aggregate	68.68%	30.32%	1.01 %
Join	3.46%	0.16%	91.18%

Table 4: Clustering Predicted Class to Query Class.

class classification model accuracy listed in Table 3 correctly classified 78% of the queries. The clustering model cluster vs. query class listed in Table 4 provided 70% accuracy.

Classification. For classification we used logistic regression[4], to build a classification model using the features selected in Table 2. The classification model had an overall accuracy of 77.98% with mean absolute error of 20.91%. The per query class accuracy for this classification model is listed in Table 3. The model is able to classify *select* and *join* fairly accurately, while it correctly classifies only 64.4% of *aggregate* queries. Noticeable is the very low false positive rate of join queries, at 1.7%.

Clustering. Clustering models group data in such a way that objects in the same group are more similar to each other than to those in other groups. We used Gaussian mixture models using Expectation maximization (EM) algorithm for clustering. EM finds clusters by determining a mixture of Gaussians that fit a given data set. Each Gaussian has an associated mean and covariance matrix. We chose EM over simple k-means because we wanted to use distribution based modeling as opposed to centroid based clustering where we divulge the number of clusters. While EM takes more iterations, it is good at handling outliers and has better clustering results.

The clustering was done in the classes to cluster evaluation mode. In this mode, the class attribute is ignored when generating the cluster from the underlying data. During the test phases the class labels are assigned to the cluster based on the majority value of class attribute in each cluster, then error and accuracy are computed based on this assignment. The resulting cluster-to-query class mapping is listed in Table 4. We assume that assigned clusters 0, 1 and 2 correspond to *select*, *aggregate* and *join*, respectively. Around 85% of *select* and around 91% of the *join* queries are assigned to the correct cluster. However, only 30% of *aggregate* queries are assigned to the correct cluster, with most aggregates assigned to the *select* cluster. Only 2% of *select* queries and 0.16% of *joins* are assigned to the *aggregate* cluster, indicating that this cluster is much smaller than the others.

6.2 Discussion

Feature Extraction. In our approach, we summarized the sequence data collected from running queries to summary statistics, such as mean, standard deviation, and maximum. Classification and clustering models built from the summarized data had 70% or higher accuracy. The summarization technique was used to build single dimensional feature vectors required by Weka. However, this technique resulted in loss of information provided to the machine learning algorithm. *Join* class queries were identified clearly, *select* and *aggregate* query classes showed significant overlap and lower accuracy in the classifiers. We suspect using the entire sequence data for learning, rather than using the summarization technique would provide higher accuracy. We also average values for summary metrics from the two workers. This should also have resulted loss of accuracy, but models built without coalescing the results from the two workers, *e.g.* by treating each worker trace as a separate record of query execution, showed similar accuracy.

Query Duration. The first set of experiments were run with a 2GB dataset. In this setup some of the query runtimes were less than a second. The models built from this data resulted in very low accuracy in classification and unsupervised clustering was only able to identify two cluster of query classes. With a second set of experiments we increased the dataset to 10GB, which added longer running queries. The feature extraction from these experiments resulted in features which were closer to the set of handpicked features and models built from this data also provided higher accuracy.

Join anomaly. Analysis of the data exposed an anomaly with *join* query processing on Myria. As can be seen in Figure 1, Myria displays a different signature where joining the fact table with dimension table 3, when compared to joining the fact table with dimension table 1. Both data placement and query plans for all joins are similar for both dimension tables. An analysis of *joins* with multiple dimension tables also reveals a similar pattern, where work is not distributed evenly across workers for the duration of the query execution. This indicates an inefficiency in *join* processing in certain cases. Further investigation is needed to understand the underlying cause for for this behavior.

7. CONCLUSION

In this paper, we explore the feasibility of a tool that is able to automatically capture and expose design decisions to users, providing rich information about the operations of the system in question using only low-level system measurements. We focus on the query execution layer of Myria, a parallel shared nothing Big Data management system from the University of Washington. Based on intial exploration the results are promising. Models built from a relatively small dataset of measurements from a fixed 10GB sized workload and fixed cluster size were able to identify query classes with some accuracy. *Join* class queries were clearly distinguishable from *select* and *aggregate* queries. In addition we discovered anomalous behavior in Myria for joining some of the dimension tables, uncovering inefficiency that was unknown to Myria developers. All of this was done without any instrumentation or code profiling. The dataset for short queries did not contain long enough sequences to give use significant measurements given we collect data per second granularity. This has implications on the workload used to understand the system.

For a fully fledged study, more data is needed. Future work will include varying a number of experiment variables. For the workload, we plan to vary the size of input tables, going beyond the 10GB dataset, and getting a richer set of more varied queries. Changing the input tables entirely with a different workload will tell us how workload-specific our approach is. An interesting challenge is to engineer queries such that they are guaranteed to produce certain system behaviors, which can give us labeled training data that not only classifies executions by query classes, but by which design decisions occur in those records. We would also like to vary configuration parameters, such as number of workers and different amounts of resources. Finally, we would like to apply this same methodology to a different system to further see how general our approach is.

8. REFERENCES

[1] J. Duggan, U. Çetintemel, O. Papaemmanouil, and E. Upfal. Performance prediction for concurrent database workloads. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 337–348, 2011.

[2] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, Nov. 2009.

[3] D. Halperin, V. T. de Almeida, L. L. Choo, S. Chu, P. Koutris, D. Moritz, J. Ortiz, V. Ruamviboonsuk, J. Wang, A. Whitaker, S. Xu, M. Balazinska, B. Howe, and D. Suciu. Demonstration of the Myria big data management service. In *SIGMOD*, pages 881–884, 2014.

[4] P. Komarek. Making logistic regression a core data mining tool: A practical investigation of accuracy, speed, and simplicity. In *Institute, Carnegie Mellon University*, pages 685–688, 2005.

[5] H.-P. Kriegel, P. Kröger, and A. Zimek. Clustering high-dimensional data: A survey on subspace clustering, pattern-based clustering, and correlation clustering. *ACM Trans. Knowl. Discov. Data*, 3(1):1:1–1:58, Mar. 2009.

[6] B. Mozafari, C. Curino, A. Jindal, and S. Madden. Performance and resource modeling in highly-concurrent oltp workloads. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 301–312, New York, NY, USA, 2013. ACM.

[7] B. Mozafari, C. Curino, and S. Madden. Dbseer: Resource and performance prediction for building a next generation database cloud. In *CIDR*, 2013.

[8] X. Pan, J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan. Ganesha: Blackbox diagnosis of mapreduce systems. *SIGMETRICS Perform. Eval. Rev.*, 37(3):8–13, Jan. 2010.

[9] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.