Microlearner: A fine-grained Learning Optimizer for Big Data Workloads at Microsoft

Alekh Jindal
Gray Systems Lab
Microsoft
alekh.jindal@microsoft.com

Shi Qiao
Azure Data
Microsoft
shqiao@microsoft.com

Rathijit Sen
Gray Systems Lab
Microsoft
rathijit.sen@microsoft.com

Hiren Patel
Azure Data
Microsoft
hirenp@microsoft.com

***Abstract*—Big data systems have become increasingly complex making the job of a query optimizer incredibly difficult. This is due to more complicated decision making, more complex query plans seen, and more tedious objective functions in cloud-based big data workloads. As a result, production cloud query optimizers are often far from optimal. In this paper, we describe building a learning query optimizer for big data workloads at Microsoft. We make four major contributions. First, we describe the challenges in cloud query optimizers based on our observations from the big data workloads at Microsoft. Second, we discuss what makes machine learning an attractive approach to aid the big data query optimizers in decision making. Third, we present *Microlearner*, a practical approach to characterize large cloud workloads into smaller subsets and build *micromodels* over each subset to tame the complexity of big data workloads And finally, we describe the productization of Microlearner, using learned cardinality as a concrete example, via performance results over very large production workloads and illustrating the various challenges involved in deployment.**

I. Introduction

Query optimization has increasingly become a challenge in cloud based big data systems. This is due to three main reasons. First, query optimization has become much more complex in big data systems with too many decisions to make, including things like efficient data distribution, data movement, and resource allocation, e.g., number of containers, size of containers, etc. Second, big data workloads are getting increasingly more complex, with larger DAG of operators, more business logic encoded into custom user defined operators, and the presence of advanced operations such as machine learning, linear algebra, etc. Third, even the optimization functions have become more complex in cloud environments where multiple objectives such as optimizing data pipelines, end to end costs, resource consumption, and dollar costs are important to cloud consumers and therefore one or more need to be considered at the same time. Altogether, query optimization has become a nightmare in big data systems and current optimizers result in decisions that are far from optimal.

Interestingly, there is a recent trend of replacing components of a database system with machine learning models [1], and even replace the entire query optimizer with a learned one [2], Indeed, this is a very promising direction for taming the above complexities in cloud-based big data systems, i.e, instead of building a general purpose big data system for all possible workload scenarios, one could train the query optimizer on the characteristics of the cloud workloads that is seen at hand. Such a learning optimizer can capture the complex cloud system behavior into learned models, scale the number of decisions by simply learning more models, optimize for complex objectives by employing different training techniques, and help make specialized decisions for different instances of workloads. However, training such an optimizer is not trivial for big data due to the humongous search space and the more complex decision making, i.e., while approaches over traditional databases have focused on cardinality estimation [3], join ordering [4], and plan search [5], there are many more important decisions to consider in the big data systems.

Motivated by these observations, we look into the SCOPE [6], [7] query engine that is used for running production big data workloads across the whole of Microsoft, including business units such as Bing, Windows, Office, Xbox, etc. Instead of making the optimization decisions from scratch every time, our goal is to learn from the behavior of past SCOPE workloads in order to improve its decisions in future queries, i.e., adding a feedback loop that continuously rectifies and guides the decision making. Luckily, such a design becomes possible with the availability of massive query workloads in modern clouds, both due to the popularity of cloud infrastructures and the rise of serverless query processing capabilities [8]–[10] that have put the onus of workload optimization on the cloud provider anyways [11]. Adding workload-awareness into the query optimizer is therefore a win-win both for the customers as well as the cloud providers.

Thus, in this paper, we present *Microlearner*, a fine-grained learning optimizer for big data workloads at Microsoft. We start by showing a detailed analysis over production workloads on why query optimization is turning out to be a big pain in big data (Section II). Then, we argue why machine learning is a helpful tool for big data given the shared nature of big data workloads, the changes in cloud workloads over time, and the current state of optimizer decisions (Section III). We present a workload characterization-based learning approach that trains *micromodels* over different subsets of the workload and puts them together (Section IV). We describe various aspects of Microlearner, including scalable model training, regression validation, model management, in-optimizer scoring, and overall developer and customer experience. Finally, we describe our journey in productizing the first version of

Microlearner, using learned cardinalities as a concrete scenario and show the significant performance gains achieved in production (Section V). We conclude by identifying the open challenges and discussing the lessons learned (Section VI).

II. Optimizing Big Data Queries

The past decade has seen a tremendous interest in large-scale data processing across industry. At Microsoft, the typical scenarios include building business critical pipelines such as advertiser feedback loop, index builder, and relevance/ranking algorithms for Bing; analyzing user experience telemetry for Office, Windows or Xbox; and gathering recommendations for products like Windows and Xbox. To address these needs, a first-party big data analytics platform has been developed at Microsoft, which makes it possible to store data at exabyte scale [12] and process in a serverless [13] form factor, using SCOPE [6], [7] as the query processing workhorse. SCOPE workloads consist of analytics jobs from almost every single business unit at Microsoft, processing hundreds of thousands of jobs per days, across hundreds of thousands of machines, with individual jobs that can consume tens of petabytes of data (and produce similar volumes of data) by running millions of tasks in parallel [14]. While much of the focus so far has been on handling the scale and complexity challenges, increasingly there is a pressing need to improve the system efficiency and reduce operational costs in these big data infrastructures. As a result, there is a renewed focus on query optimization over big data. Unfortunately, it turns out that optimizing big data queries is challenging due to several reasons. Below we describe some of these challenges based on our observations from one day's worth of SCOPE query workloads at Microsoft, consisting of hundreds of thousands of jobs from thousands of users and hundreds of business units. We believe that the challenges discussed below are also applicable elsewhere.

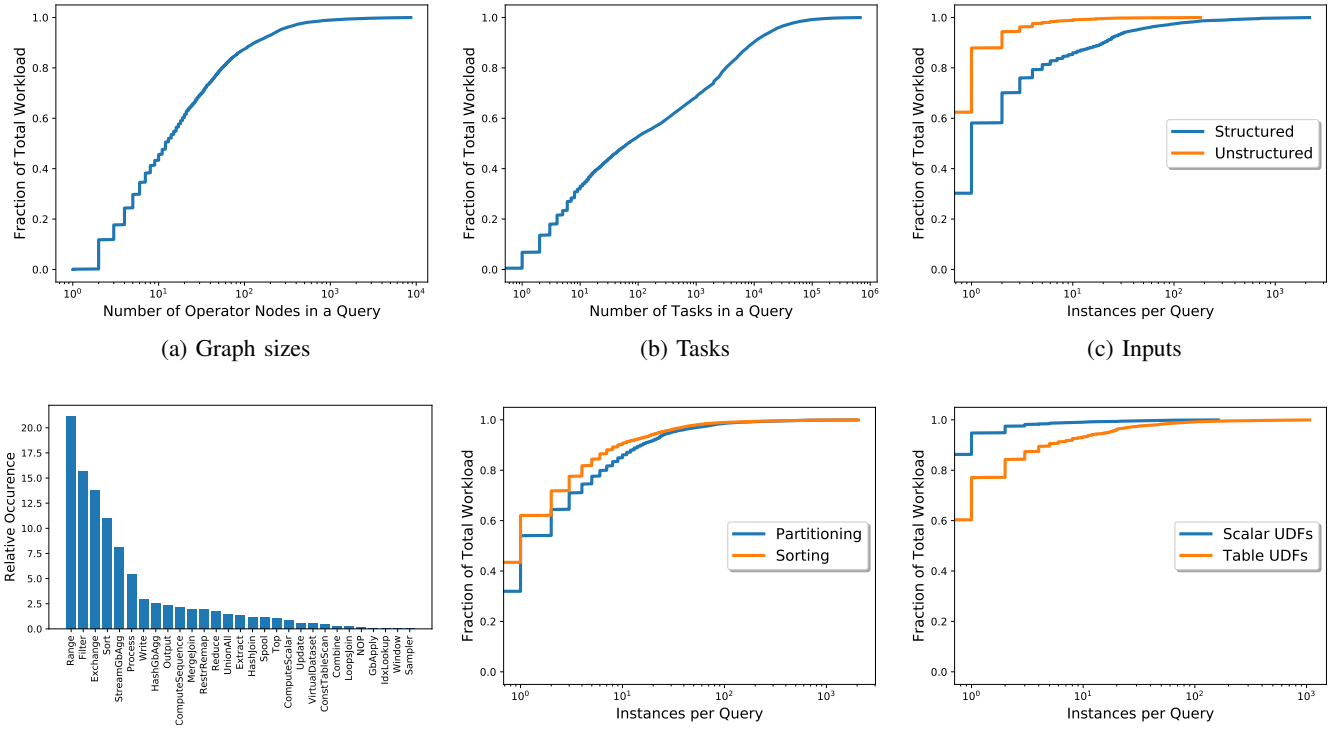
Query graph sizes. Figure 1a shows the cumulative distribution of the size of SCOPE job graphs. We see that while 40% of the jobs have within 10 operator nodes, the remaining can have much larger DAGs — 20% having more than 50 operators, 13% having more than 100 operators, and 3% having more than 500 operators! Thus, we see that compared to the traditional databases, big data systems have to deal with a tail of very large queries, which could also be business critical. This is because SCOPE users author the SCOPE scripts by writing a sequence of statements, in a data flow style, with multiple inputs and outputs that together get compiled into a single giant DAG. While this makes it easier for the users to express their business logic, larger DAGs also make the job of the query optimizer harder, since the plan search space and the query optimization complexity grows exponentially with the number of nodes. Furthermore, since big data queries run in a distributed manner, the optimizer also needs to determine the data distribution (partitioning, sorting, etc.) at each point in the query plan, which further adds to the complexity. Therefore, to produce a query plan in reasonable time, the SCOPE query optimizer employs a conservative set of transformation rules and join ordering heuristics, that may end up far from the optimal plans. Large query DAGs also render many of the optimizer estimates, such as cardinality, skew, correlation, highly inaccurate, introducing more errors in picking the physical execution plan.

Distributed execution. SCOPE jobs group operators into stages and runs each stage as multiple tasks in a distributed manner. Figure 1b shows the cumulative distribution of the number of tasks in each SCOPE job. We see that more than half of SCOPE jobs have more than 100 tasks each, with a third having more than 1000 tasks and a tenth having more than 10000 tasks! Thus, SCOPE like big data queries are resource intensive, and orchestrating and executing these large number of tasks in a reliable fashion is a challenge. It is therefore important to optimize the distributed execution by considering number of tasks, packing tasks into containers, setting the degree of parallelism at each point in the query plan, deciding the maximum degree of parallelism for the entire query, and balancing between latency and throughput.

Inputs. Figure 1c shows the cumulative distribution of the number of structured and unstructured inputs in each of the SCOPE jobs. First, note that 40% of the SCOPE jobs have unstructured inputs. Jobs with unstructured inputs are harder to optimize due to lack of schema, lack of statistics, such as cardinality, min/max, etc., as well as interesting physical properties, such as partitioning and sorting, that can aid in producing more efficient execution plans. Furthermore, note that several SCOPE jobs access multiple inputs — 15% have more than 10 structured inputs, 5% have more than 50 structured inputs, and 3% have more than 100 structured inputs, i.e., a tail of very wide and often critical jobs. Large number of inputs compound the error due to many different correlations that are very difficult to mine for petabytes of big data. Likewise, 10% of jobs have multiple unstructured inputs, again compounding the errors due to missing statistics.

Operators. Figure 1d shows the relative occurrence of different operators in the SCOPE workloads. Note that the shuffle operator (called *Exchange* in SCOPE) is the second most occurring operator after the scan operator. This is interesting because shuffle is also one of the most expensive operators in big data processing [15], and so it is crucial to optimize for that. Furthermore, shuffles are almost five times more frequent than joins (merge, hash, and loop join taken together). In fact, even sorts and user defined operators are both three time more frequent, and often more expensive, than the join operations. Therefore, in contrast to traditional database wisdom to optimize for joins, there are other more important operators to consider in optimizing big data processing. We dig further into the data movement and user defined operators below.

Data movement. Figure 1e shows the cumulative distribution of data movement (shuffle and sort) in each of the SCOPE jobs. We see that two third of the jobs involve data shuffle, with half of jobs having multiple shuffles, 28% having 5 or more shuffles, and 17% having 10 or more shuffles (5% even having 50 or more shuffles per job!). Likewise, 56% of the jobs have sort operations, with 20% having 5 or more sort operations, and 12% having 10 or more sort operations. Thus,



(a) Graph sizes (b) Tasks (c) Inputs

(d) Operator frequencies (e) Data movement (f) User defined operators

Fig. 1: Analyzing the query optimization challenges from production big data query workloads at Microsoft.

data movement is prevalent in SCOPE-like big data workloads, and it is important to optimize these operations. Typical techniques include creating the right physical designs, e.g., partitioning, sorting, etc., for intermediate and final outputs, tuning the partition counts for balancing the shuffle costs with the parallelism benefits and considering plans that minimize the data movement in the first place.

User defined operators. Finally, Figure 1f shows the cumulative distribution of user defined operators (UDOs) in each of the SCOPE jobs. We see 15% of the jobs have scalar UDOs, with 8% having multiple scalar UDOs. Likewise, 40% of the jobs having table UDOs, with 20% having multiple table UDOs — 10% having 5 or more and 5% having even 20 or more table UDOs. The reason for such massive use of UDOs in SCOPE workload is because users often want to express their business logic using a mix declarative and imperative code fragments. Unfortunately, however, UDOs appear as black boxes in the query plans and are very difficult for the optimizer to reason about: it is hard to estimate cardinality or cost of the UDOs, and even hard to apply simple transformation rules such as selection or projection push down.

In summary, query optimization is much harder in big data systems, making the job of systems developers difficult. The problem becomes worse in cloud services, where users neither have the control nor the expertise to tune the query optimizer for their workloads [11]. Thus, we need automatic techniques for improving query optimization in big data and below we discuss why machine learning is a suitable approach.

III. Why Machine Learning?

We saw that query optimization is a much harder problem in big data. In this section, we discuss what makes machine learning attractive for approaching this problem.

A. Shared Cloud Workloads

First, big data systems are typically deployed in cloud infrastructures where massive volumes of workload traces become centrally visible to the cloud provider. This is a shift from traditional on-premise databases where the fragmented workloads are locked in customer infrastructures. The centralized nature of the infrastructure also incentivizes the cloud engineers to continually add more instrumentation for better tracking, monitoring, and troubleshooting. To illustrate, the SCOPE analytics platform collects job-level metadata (users, accounts, queuing details, start/submit/end times, etc.), plan-level data (logical plan, physical plan, stage graph, optimizer estimates), runtime statistics (various operator- and stage-wise observables), task-level logs, and machines counters — together collecting a rich repository of several TBs of workload metadata per day that could be used for training sophisticated machine learning models. The shared nature of big data infrastructure also helps train models over a diverse set of workload patterns that could be identified across different customers and accounts.

B. Workload Changes

Big data workloads are also constantly evolving over time. To analyze the changes, we consider a 7-month trace of

(a) Workload Size (b) Job Size (c) Job Duration (d) Job Tasks

(e) Cluster Load (f) Input Size (g) Data Movement (h) User Defined Operators

Fig. 2: Workload changes in one of the production big data clusters at Microsoft over 18.2 million jobs from a 7-month period.

SCOPE workload from one of the production clusters. This trace consists of a total of 18.2 million jobs, processing a cumulative input size of 40 Exabytes, and having a cumulative latency of 3 million hours.

Figure 2 illustrates aggregated daily changes over several metrics. First of all, the number of SCOPE jobs grow by 24% over the 7-month period (Figure 2a). The jobs are also 27% larger jobs in terms of the average number of operators per job (Figure 2b). However, the job runtime changes are staggered because while the inputs sizes and job complexity grew over time, there are also newer efficiency features added to the engine or the underlying platform; as a result, the average latency climbs by almost 90% before coming down to just a 30% increase (Figure 2c). There are also 69% more number of tasks per job on average, indicating more distributed processing (Figure 2d). The queuing time also sees changing behavior as cluster load and capacity changes over time, still the average queuing time is up by 80% indicating a very different cluster load conditions (Figure 2e). Barring a few seasonalities, the input sizes per job shows an upward trend with an increase of around 50% from start to end (Figure 2f). While the data movement has spikes in the closer intervals, interestingly, it remains relatively flatter in the longer trend (Figure 2g), We also see a rather unsuspecting change of around 30% in the number of user defined operators per job (Figure 2h). This could be triggered by better support for custom code, adding more programming languages like Python, and UDO efficiency features.

We see that big data workloads are constantly changing and the query optimizer need to cope up with these changes when making the optimization decisions. This is non-trivial unless the optimizer can constantly update its models by taking into account the past changes – something not really possible to do manually (e.g., updating a cost model could take several months if not years, and workloads would have changed heavily by then). Machine learning can help the query optimizer to constantly adapt by training over complex workload changes and even anticipate the future ones.

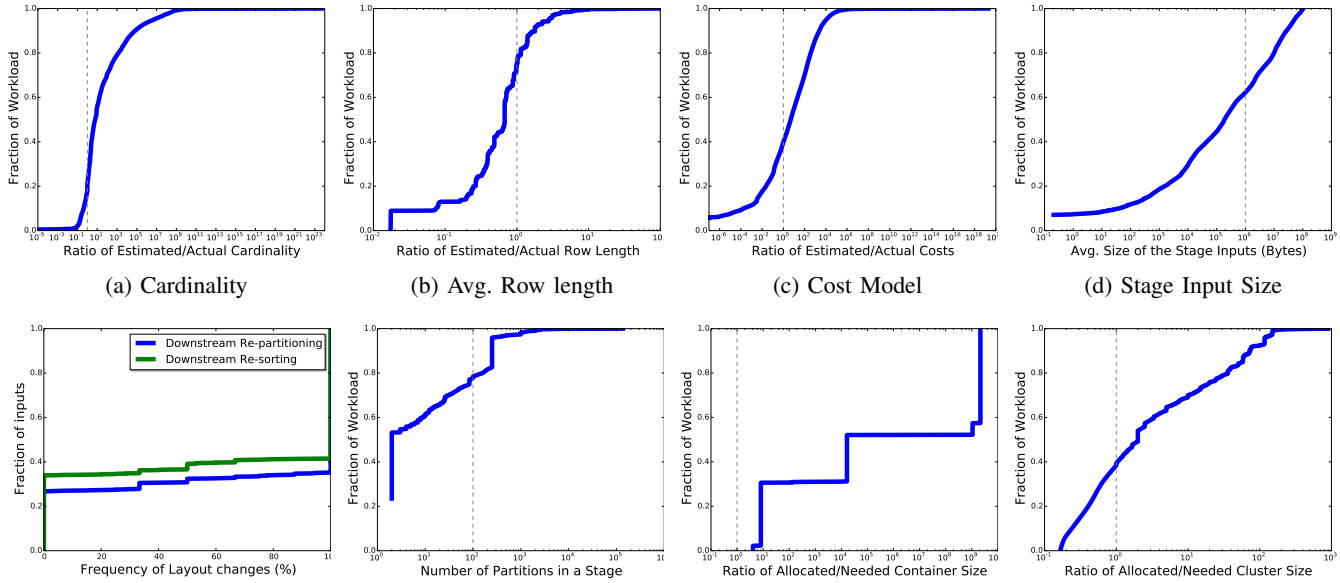
C. Current State of Optimization Decisions

We now discuss the current state of several heuristics and decisions in SCOPE query optimizer. We consider 1-day workload from the same cluster as analyzed above.

Cardinality. Cardinality is a key statistics for query optimization and there has been a lot of recent attention on learning cardinality models. Figure 3a shows the ratio of estimated and actual cardinalities in the SCOPE workloads. SCOPE like big data query optimizers tend to over-estimate the cardinality in order to avoid having straggler nodes or even failures. The downside though is that the optimizer may not only end up choosing sub-optimal plans, it may also over-partition the intermediate data, thus creating a large number of very small partitions. Learning cardinalities is challenging in big data systems due to very large query graphs that make estimation errors propagate exponentially and presence of large number of user defined operators that are hard to reason about.

Row Length. Row length is another piece of information, other than cardinality, that turns out to be very important. It is used by the SCOPE runtime to compute memory allocation for each stage. Figure 3b shows the ratio of estimated and actual row lengths in our analyzed workload. In contrast to cardinality, row length is mostly underestimated since it is mostly set to a statically determined constant value.

Cost Model. Cost modeling is a well-known problem and various prior works have proposed to learn performance models. Figure 3c shows the ratio of estimated and actual query latencies. Note that, in contrast to the popular belief, fixing



(a) Cardinality (b) Avg. Row length (c) Cost Model (d) Stage Input Size

(e) Pipeline Design (f) Partitions (g) Container Size (h) Cluster Size

Fig. 3: Current state of some of the decision making in SCOPE, illustrating the opportunities where the learning optimizer could make a difference.

cardinality estimates does not fix the cost model inaccuracies. This is due to a large number of other moving parts in the systems, e.g., the way the query is distributed, the amount of resources allocated, etc., not to mention the presence of large volumes of custom user code whose cost is very hard to model.

Pod Aggregate. SCOPE job manager can apply rack-level *combiners* (from the Hadoop MapReduce terminology) or pre-aggregations, and the optimizer can provide hints for doing that. Such an aggregation is useful when a stage is processing small amounts of data across a large set of machines. Figure 3d shows the stage input sizes. We can see that large number of stages process smaller amounts of data and could be ideal candidates for pre-aggregation. However, estimating the stage input sizes so that the pre-aggregation can be injected judiciously turns out to be non-trivial.

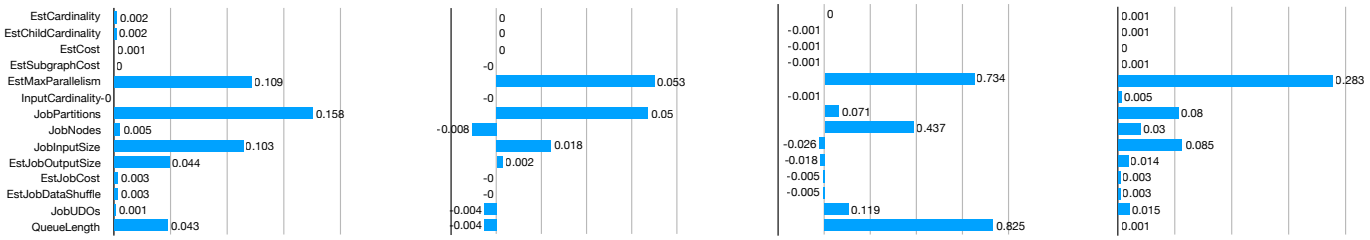
Pipeline Physical Design. Figure 3e shows percentage of times an input data stream is re-partitioned or re-sorted in a downstream query. We can see that almost 60% of the inputs are re-partitioned or re-sorted every single time they are accessed in the downstream query. This is because there are thousands of SCOPE developers spread across multiple teams and locations, and it is hard to optimize the producer-consumer relationships. Rather, providing feedback from past workloads can help solve this problem. However, the challenge is that past workloads, and inputs in particular, keep on evolving. Therefore, the physical design learnings need to consider things like the time when the design must be scheduled, the variations in the design, e.g., number of partitions, sort orders, etc., and the time to live for the designs.

Partition Count. Cloud query optimizers need to decide the data parallelism, i.e., the number of partitions for each stage in the query execution plan. However, beyond a point, increasingly parallelism does not necessarily improve performance [16]. Figure 3f shows the number of partitions in different stages of our workload. We can see that 25% have more than 100 partitions and there is a long tail with up to a million partitions. These are obvious candidates to reduce the data parallelism. Apart from picking the operator implementation, partition count is also useful when interfacing with a disaggregated blob store that is not optimized for analytical workloads. In such a scenario, providing partition count hints can help create large block sizes and provide better locality compared to other cloud workloads supported by the same blob store.

Container Size. Determining the right size of containers to be used for processing is increasingly getting more and more important, especially with increasing heterogeneity in SCOPE like big data workloads. Figure 3g shows the current state in terms of the ratio of allocated and needed memory sizes of containers. We can see that in almost all cases, we allocated much large containers than needed. Thus, there is a significant opportunity to learn memory models from the past workload and tune down the containers to really required sizes.

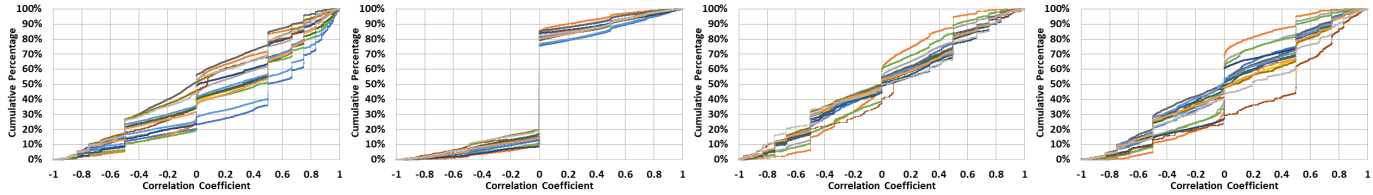
Cluster Size. Finally, there is a new trend for serverless data processing. For instance, SCOPE is a job service where each SCOPE job is allocated a maximum number of containers that could be used for that job. The current practice is for the users to provide this configuration for every SCOPE job. Figure 3h shows the allocated number of containers (as specified by the users) and the maximum number of containers needed. We can see that the allocation is all over the place, with 60% of the workload ending up over-allocating. Again, building models for estimating the cluster size could help predict the cluster much more accurately.

Thus, we see that it is really hard to get the query optimizer estimates and decisions right over complex big data workloads.



(a) Cardinality (b) Cost (c) Container Size (d) Cluster Size

Fig. 4: Pearson correlation for four estimations, namely cardinality, cost, peak memory, and maximum parallelism.



(a) Cardinality (b) Cost (c) Container Size (d) Cluster Size

Fig. 5: Pearson correlation distributions for four above estimations when using the micromodel approach.

IV. MICROLEARNER

We saw how query optimization is a much harder problem in big data, both due to the large space of workloads and the large number of decisions to make. We also saw that machine learning becomes attractive for this problem due to the presence of shared cloud workloads, where one can learn across several users, the constantly changing nature of cloud workloads that is very hard to cope up with using manually crafted query optimizers, and the difficult state of current optimizer heuristics and decisions that are simply very hard to get right due to the complexity of the problem.

One size does not fit all. Given the massive workloads available in modern services, it is tempting to simply feed them into a machine learning library. Unfortunately, it turns out that learning machine learning models that capture the global behavior of the entire workload is incredibly difficult, if not infeasible. To illustrate, Figure 4 shows the global correlations of the actual values of four key metrics (cardinality, cost, container size, and cluster size) estimated by the query engine with several of the compile-time features. We can see that single variable correlations are very low, making it hard for data scientists to featurize the complex big data workload characteristics using traditional machine learning. Apart from complexity, big data query workloads are often sparsely populated dense clusters, i.e., even though there are similar queries, it is still highly sparse overall. This means that even the large cloud workloads are far from sufficient to identify hidden representations and train deep learning models.

Micromodels. We propose to characterize big data workloads into fine-grained subsets and learn separate models, called *micromodel*, for each of those subsets. Figure 5 shows the correlations with micromodels when the workload is divided into smaller specialized subsets. We can see that the correlation increases substantially for a large number of subsets. Thus, micromodels make it possible to learn accurate models over complex cloud workloads. Micromodels are further helpful for scalable model training by training different micromodels in parallel, creating lightweight models that are easier to score within the query optimizer, and even isolating performance regressions by selectively enabling the good micromodels.

System Architecture. Figure 6 shows the Microlearner architecture that consists of optimizer extensions for additional telemetry to help characterize the workloads and apply targeted feedback to specific types of workload (center box), artifacts repository to collect telemetry, often in different stages, and derived datasets (right box), analysis pipeline to simplify and scale the model training process and to build confidence for productization (bottom layer), model serving layer to facilitate low latency model lookup and to perform diagnostics and control (left box), tools for a productive operational and data science user experience (top layer).

Applications. Over the last couple of years, we have used Microlearner to build several applications, including fine-grained models for learning cardinality estimates [3], robust and resource-aware cost models [17], models for predicting peak parallelism of analytical jobs [18], models to predict reuse opportunities for multi-query optimization [19], predicting checkpoints for job resiliency or system efficiency, predicting the downstream behavior in a data pipeline to optimize the end to end pipeline, predicting when to combine intermediate data (early aggregation), predicting peak memory consumption to improve the allocation at runtime, and predicting average row length to correctly estimate the intermediate size for data movement. Likewise, there are other data properties (e.g., correlations, skew), cost models (stage-wise, job-wise), optimizer rule hints (to better navigate the search space), and resource optimization decisions (e.g., container type) that could be learned. Thus, the Microlearner architectures allows a large class of applications to improve the performance and efficiency of cloud query engines.

In the rest of this section, we describe the core platform that has made it easy to quickly build the above applications.

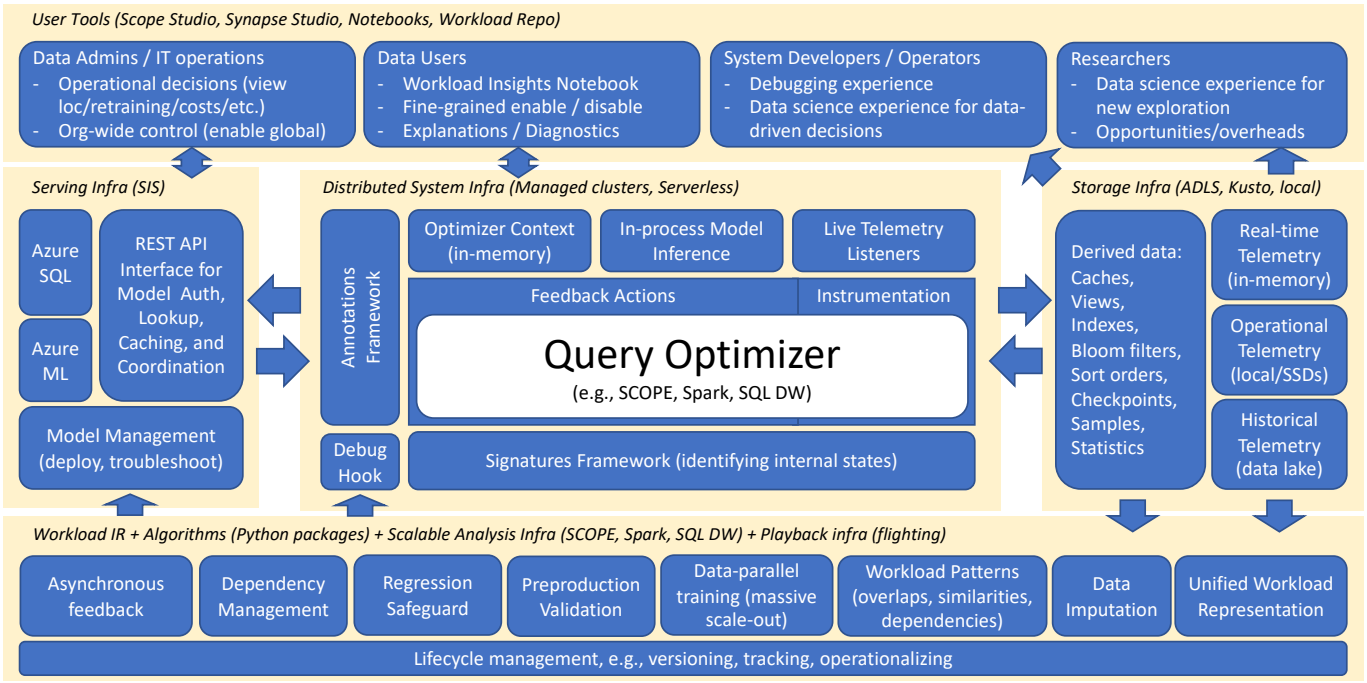


Fig. 6: The detailed set of components in the Microlearner platform.

A. Workload Characterization

Microlearner captures the traits of complex cloud workload into a hierarchy of subsets, where each subset has a unique identifier called *signature*. The most straightforward characterization is to consider each query in a workload as different subset. Then, the signature for each query could be the query name or a hash of the query string. However, queries could be further characterized using the query plans, e.g., the input plan, the logical plan, the physical plan, or the execution plan, and then the sub-plans in each of the query plan. The plan (the or sub-plan) could be represented by a hash of plan (sub-plan) string, or the expression id of the plan (sub-plan) from the optimizer's memo (e.g., semantic hash in Spark Catalyst). We can further characterize sub-plans by first considering the root operator of the sub-plan, then the set of inputs at the leaf level, and then the operator counts in the sub-plan. Likewise, we can add arbitrary number of features from the sub-plan to make the characterization narrower, until we have sub-plans with the same structure and the same parameters. As we go down this characterization tree, the workload is divided into smaller and more homogeneous subsets that are easier to train models on. The depth of the characterization tree is chosen such that final subsets are small enough to learn accurate models on the given workload. To illustrate, Figure 5 shows the CDF of correlations for different workload subsets. We can see that correlations change significantly with subset granularity, with significantly higher correlations in many of the subsets. Figure 7 shows how the workload gets characterized differently based on the subset strategy, e.g., characterization using exact query subgraph can create many fine-grained subsets (low frequency), while characterization based on operators create more coarse-grained subsets (high frequency); others are somewhere in between.

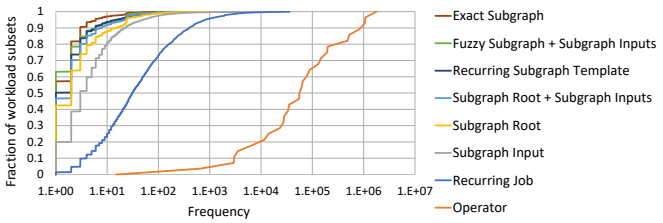


Fig. 7: Frequency distributions of different workload subsets.

B. Scalable Model Training

The above workload characterization is helpful in scaling out the training by learning micromodels for different workload subsets independently in parallel. As a result, Microlearner can scale to very large workloads by leveraging the massive parallelism in cloud systems. We further represent the query workload as a flat denormalized table consisting of both compile-time and run-time features for all operators in all queries in the workload [11]. This tabular representation coupled with the fact that we train micromodels on smaller workload subsets that are more homogeneous makes linear models very attractive, both for accuracy as well as performance and maintainability. In fact, popular big data systems, such as SCOPE, Spark, and SQL DW already support custom Python UDFs where popular machine learning libraries such as Scikit-learn [20] could be used to train micromodels with existing data infrastructure[1], without requiring separate training infrastructure or expensive hardware. This is convenient since all analysis and training can run in a single environment.

Microlearner hash partitions the training data on signatures and distributes them over hundreds of containers for parallel processing using SCOPE. Microlearner further provides the

[1]Could be the same engine that we are trying to optimize in the first place.

flexibility to change the training window, from days to weeks to months, and the training granularity, from the strictest subplans to the generalized operators. It also allows to combine models from different granularity into a hierarchy of micro-models [21], [22]. For retraining, it is easier to simply replace the specific micromodels with their newer versions.

C. Regression Validation

One of the biggest challenge in deploying micromodels is to avoid performance regression, i.e., learned models can lead to a detrimental system behavior. This could be due to a number of factors, including large prediction errors, high sensitivity to workload changes, lack of coverage on portions of the workload due to the fine-grained nature of model training, or the model impacts heuristics whose inaccuracies were compensated for in other parts of the system (e.g., poor cardinalities are sometimes compensated in the cost models). Therefore, it is crucial to build the confidence for production. Fortunately, due to their fine-grained nature, micromodels can be selectively filtered to reduce the chances of performance regressions. Microlearner provides multiple levels of checks to do this filtering. First of all, micromodels can be filtered based on their training and cross-validation errors, e.g., we can define the thresholds for minimum baseline and maximum training errors. We then recompile the jobs in the training data set using the filtered models and separate out the models that are applicable to jobs with query plan changes (in structure or estimators like execution cost). Models that only impact jobs with no plan changes are discarded since they do not improve anything. We validate the set of filtered models against a workload from a subsequent time period to consider minor workload changes and filter models that are sensitive to them. Thereafter, we run preproduction experiments on the production jobs that are going to be impacted by the validated models. In case a large number of jobs are going to be impacted, we deduplicate similar jobs and take a sample of them. We compare the preproduction performance of jobs, with and without the validated models, and consider several factors (latency change, total number of containers, total processing time) to determine whether or not we have a regression. We add the models that do not contribute to any regressions to an accept list and deploy them to production. While we have a functional approach for regression validation in Microlearner, there is a lot of room for improvement, especially better isolation of models causing plan changes, better sampling for reduced preproduction experimentation costs, and more accurate regression identifiers, considering cluster variance. These will be a part of future work.

D. Model Management & Lookup

Model training in Microlearner is scheduled periodically based on factors such as experimentally derived retraining intervals, data or concept drift, changes in model coverage based on the number of model lookup calls, or other external factors, e.g., new query engine release or customer specified training intervals. We serialize the resulting micromodels into a feedback file and upload it asynchronously to the serving infrastructure. For linear models, serialization is simply into key-value pairs of features and their weights. Each model is also accompanied by its signature (identifying the workload subset it was trained on) and the model type (e.g., cardinality, cost, memory, parallelism, etc.)

Microlearner keeps track of the following model metadata. First of all, we keep track of the query engine version from where the training data came from. This is important because different query engine versions could evolve various estimators, query optimization logic, or even the runtime performance characteristics. Then, we track the version for the analysis and training scripts to differentiate changes in preprocessing steps and the training logic. We also track the parameters used for training, including the customer accounts whose data is used for training, the date range over which the models are trained, and any other configurations used by the models. Finally, we keep track of model dependencies, i.e., whether predictions from one model might be used as features for another model. The above tracking is helpful for debugging, explainability, and purging models with specific lineage, such as for GDPR scenarios [23]. All tracking is done using the MLflow APIs in Azure Machine Learning [24].

E. Model Lookup & In-Optimizer Inference

Micromodels are served over a REST interface to the query engine. The difference though is that instead of serving predictions, the serving layer serves the model strings that could be scored in-process within the query optimizer. Furthermore, all models strings for a given query are batched together in a single call to reduce the communication overhead. As a result, Microlearner can load a large number of models with very low latency (10s of milliseconds) into the query optimizer. We load the micromodels as annotations to the query optimizer. These annotations are indexed by the signatures and contains a list of models, along with their types, for each signature. These query annotations are preserved in the optimizer context and available throughout query optimization for possible action. To take action, the optimizer performs the following steps: (1) iterate over all available group[2] signatures, (2) for each signature consider available models, (3) for any applicable model, invoke the scoring function, and (4) use the score to replace heuristics with better decisions The last step involves targeted code change for every feature where we replace hard coded constants, inaccurate heuristics, or a complex logic with a simplified inference from the micromodels. Given linear models and in-process scoring within the query optimizer, the code changes can even speed up complex code paths by taking alternate paths with faster model inference, e.g., predicting cardinalities over complex subexpressions could be faster with linear models compared to applying complex heuristics.

F. Monitoring & Incident Management

Whenever a micromodel is applied in the query optimizer, we log the usage in the plan info logs. These logs could

[2]Group of equivalent query subexpressions.

also be used in case of production incidents to quickly narrow down whether the incident is due to one or more micromodels. For quick mitigation, there are job-level flags to disable specific micromodels. If case the mitigation is confirmed, then micromodels could be disabled for specific jobs or customer accounts. Furthermore, due to the fine-grained nature of micromodels, Microlearner can also disable specific incident-causing models while still keeping the rest of them operational. In the extreme case, all micromodels (of a particular type or globally) could be disabled.

G. User Experience

Our goal is to provide a configurable, debuggable, and explainable experience to our users. Therefore, we provide several configurations to enable/disable different features, training analysis and insights to help customers build confidence (e.g., observing the model accuracy and other metrics, manually inspect and verify the baseline behavior in their jobs, and identify the KPIs to keep track of once they enable a set of micromodels), and cooked (anonymized) workload telemetry for exploratory analysis by researchers and developers to identify the next set of opportunities.

V. Productization

In this section, we describe our production experience and the challenges faced when deploying Microlearner in SCOPE. We take the example of cardinality models [3] as a concrete scenario and discuss the process of upgrading the SCOPE optimizer from default cardinality models to the learned micromodels. Below we first present micromodel training and validation on production clusters, then discuss pre-production experiments to analyze runtime performance, then show the impact in production environments, then analyze some of the production plan changes that bring significant improvements, and finally conclude with the open challenges.

A. Model Training and Validation

We trained cardinality models over a large production cluster, and considered 6 days' worth of SCOPE jobs, consisting of $\sim$564 SCOPE jobs, from the above cluster and trained cardinality models over them ($\sim 400K$ micromodels over $153M$ query subexpressions). The 95^{th} error of these learned cardinality models is just 1%, compared to 465711% for the default optimizer. We recompiled all jobs with the learned cardinality models and extracted the subset of jobs having plan changes with the learned models. These constitute roughly one-third of the total workload, with $\sim 52K$ relevant micromodels. Thereafter, we validated the relevant micromodels on 1 subsequent day of workload, consisting of $\sim 93K$ SCOPE jobs. The total training time was $\sim$6 hours and validation time was $\sim$1 hour, both using 200 containers each, and the total size of all validated micromodels was 1.5MB.

Given the training and validation results, our goal now is to identify the good micromodels that are expected to lead to better performance. After several rounds of experimentation in pre-production environment, we came up with the following heuristics to filter out the good micromodels:

(1) The average difference between the baseline cardinality estimation and the actual cardinalities must be at least 100%, i.e., current optimizer estimates must be significantly off.
(2) The maximum difference between the validated and actual cardinalities must be within 100%, i.e., validation results are up to 2x in the worst case.
(3) The average difference between the validated and actual cardinality must be within 10%, i.e., validation results are very close to actual on average.
(4) The maximum difference between validated and predicted cardinalities must be within 1%, i.e., the model must be relatively stable over time.

The above heuristics result in $\sim 10K$ high quality models that impact $\sim 20\%$ of the overall workload, which is a sizable given that learned cardinality is the most plan changing feature in SCOPE till date. Note that our heuristics are highly conservative to avoid early regressions and build customer confidence, and we expect a few rounds of retraining to stabilize the system behavior.

B. Pre-production Experiments

We group the production jobs that are applicable to the filtered set of good micromodels into *recurring pipelines*, i.e., different instances of similar SCOPE scripts that get executed periodically, and randomly pick one instance from each of these recurring pipelines. Also, given that our cardinality micromodels are sparse, i.e., they may not cover all portions of a query graph, we filter out super large sized jobs (exceeding one hour of execution time) since they are more likely to be impacted with this sparsity. This results in 1149 unique pipelines that we can run pre-production experiments on. We re-run each of these pipelines over their original inputs but redirect their output to a dummy location, similar as in prior work [3], [19], [25]. We provide the same amount of resources as in the original execution but disable opportunistic resource allocation [26] in order to be able make a fair comparison. For each pipeline, we run the randomly chosen job instance twice: once with the learned cardinality models and once with the default cardinality estimators. For each job that we ran, we report the percentage difference in job latency, total processing time, and total number of containers for the two runs of the job, with and without cardinality micromodels.

Figure 8 shows the results for pipelines exhibiting overall performance improvement (a negative change implies improvement while a positive change implies degradation). Note that pipelines behave differently on different metrics. Therefore, we consider a pipeline to improve if one of the metrics improve significantly with at most 15% latency or processing time overhead, and up to 40% more number of containers. This is to take into account runtime variance in cloud environments and to also allow limited tradeoff of one metric for another, e.g., trading more containers for lower latency. We observe from Figure 8 that the changes in latency have fewer spikes than the other two metrics, thereby suggesting that it is hard to

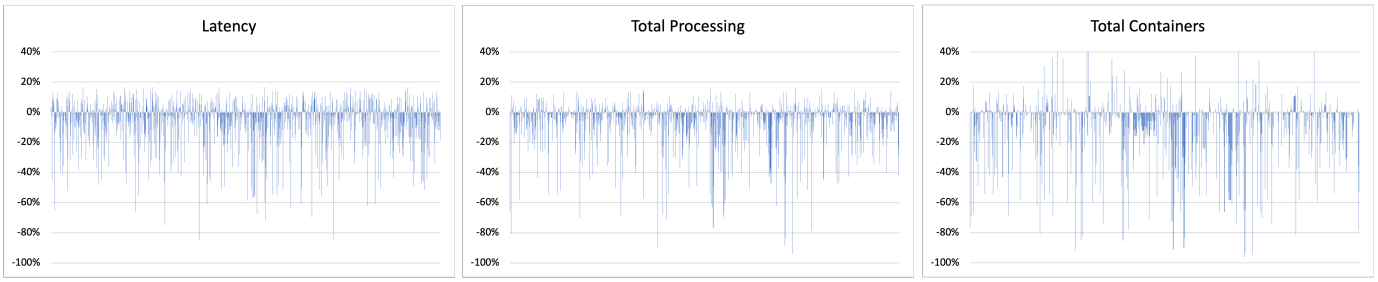


Fig. 8: Pre-production performance of learned cardinality on a large production workload of 1081 production pipelines, representing over $10s$ of thousands of production jobs in total.

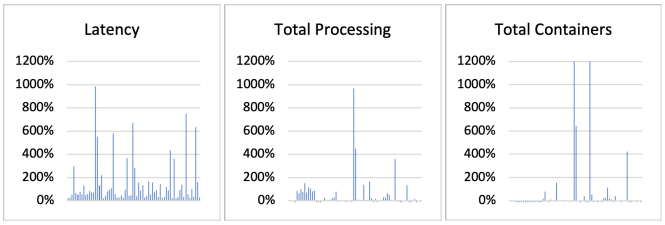


Fig. 9: Performance regressions in 68 production pipelines.

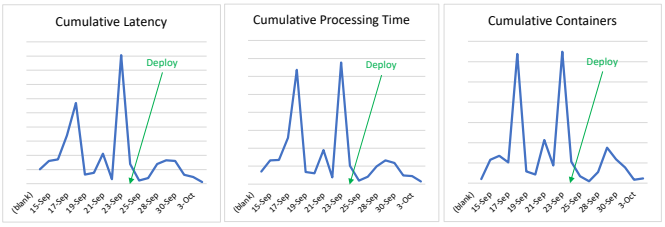


Fig. 10: Production impact for a customer since deployment.

gain massive latency improvements. The reason for this is that latency improvement in distributed query processing depends on the critical path, and better cardinalities may or may not impact that path. On the other hand, better cardinalities will most likely improve the total processing time, i.e., the sum of processing times of all containers in all paths, as evidenced by longer spikes for total processing time in Figure 8. This is highly desirable for overall cluster efficiency. Finally, the change in the number of containers is much more dramatic, more than 50% in a large number of pipelines. This is because overestimating the cardinalities leads to over-partitioning of data and hence more number of containers used for processing; imagine thousands of containers each processing few megabytes of data. This is wasteful in several ways including data movement and IO, costs to allocate the large number of containers, more load on the resource manager, and even higher chances of task failures leading to higher recovery costs. More accurate cardinalities can therefore prevent many of these issues. Overall, we see an average improvement of 6.41% in latency, 6.90% in total processing time, and 8.29% in total number of containers for the above pipelines.

Figure 9 shows performance regressions in 68 pipelines. Overall, regressions are most serious in terms of latency, which cannot regress beyond the SLAs. The total processing time and the total number of containers regress far less, although there are few spikes that are detrimental to other jobs in the cluster and for overall efficiency. Fortunately, Microlearner allows disabling the relevant micromodels for these pipelines, and thereby avoiding regressions in the actual production setting.

C. Production Deployment

Figure 10 shows the production impact when deploying cardinality micromodels for one of the customer. The green arrow shows the day when the micromodels were enabled and the three charts show the cumulative latency, the cumulative processing time, and the cumulative number of containers per day, over a 19 day period, for this customer. We can see that enabling cardinality micromodels smoothens the peaks, i.e., extreme performance behavior that is typically caused by extreme inaccuracies in cardinality estimation. Overall, since deployment, we see an average daily reduction of 69.2% in cumulative latency, 68.9% in cumulative processing time, and 71.5% in cumulative number of containers. These translate to significant efficiency gains in hundreds of millions of dollar worth of cloud infrastructure.

To the best of our knowledge, this is the first production deployment of learned cardinality models.

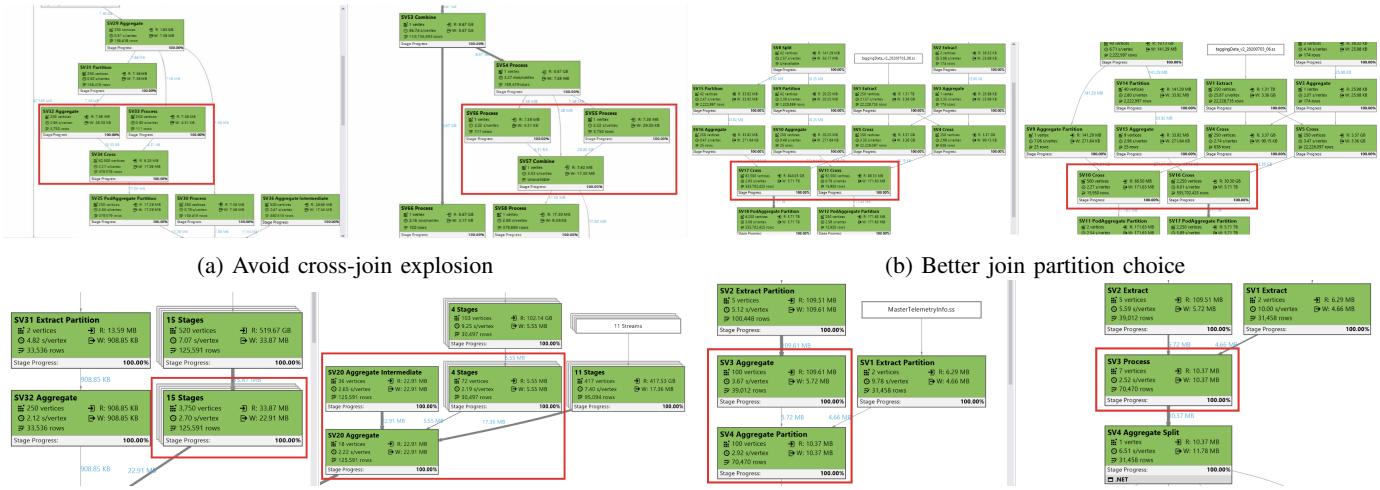
D. Case Studies

We now dig deeper into some of the cases and analyze the query plans to understand where the gains come from.

Case 1: avoiding explosion with cross-join. Figure 11a shows a cross-join operation where the inputs as well as the result are very small. However, the cardinality estimation in this case is off by several orders of magnitude which causes the cross join to explode the number of partitions, thus creating a large number of containers which process very small amount of data. After feedback, the optimizer falls back to a serial plan that saves 95% in total processing time.

Case 2: better join partition choice. Figure 11b also shows a cross-join but this cannot be converted to a serial plan since it still emits a large output. To address over-partitioning, the default optimizer adds an intermediate aggregate to both inputs, however this is not enough. Therefore, with feedback, the optimizer fixes the partition count to not over partition on one side, thus reducing 67% in total processing time.

Case 3: better grouping and partition choice in union operation. Figure 11c shows a union operator which can be implemented either as a partition aligned union all or as concatenation followed by exchange and sorting later on. Some of the inputs to the union become small after filtering operations, while others remain large. Therefore, based on



(a) Avoid cross-join explosion

(b) Better join partition choice

(c) Better grouping and partition choice

(d) Better algorithm choice

Fig. 11: Case study of a few query plan changes that lead to significant performance improvements with learned cardinality.

the feedback, the optimizer groups them in a different way and tries to minimize over partitioning, from 4000 containers processing just 30MB of data to a few hundreds, which reduces the total processing time by 73%.

Case 4: better algorithm choice for union operation. Finally, Figure 11d shows another union operation where we don't need to do paired union all, since it will increase the number of partitions, while the datasets are extremely small. Instead, using a virtual dataset to simply concatenate the inputs and performing exchange later on will produce better serial plans, leading to 95% reduction in total processing time.

E. Open Challenges

There are several open challenges when deploying micromodels to production. We list some of them below.

Imputation. First of all, the training data for Microlearner is often incomplete. While getting the query plans and the compile time estimates are trivial, it is non-trivial to link them to the actual runtime statistics. This is because a physical query plan is transformed into a set of pipelined stages, the stage graph, that runs in parallel in a distributed environment. This transformation could add or remove operators, pipeline or parallelize operations making it harder to collect fine-grained runtime statistics, and even change dynamically at runtime depending on the cluster conditions or adaptive execution. Consequently, we need more systematic approaches to deal with missing values in training data, including approaches to understand when it really matters to impute in the first place.

Training bias. Optimizers are tasked with a very large search space which inevitably leads to learning bias when learning from the past workloads. For instance, learning cardinality models could introduce bias towards the observed executions in the past. Newer inputs could in fact lead to unpredictable performance. One way to address this problem is by exploring different parts of the search space, e.g, trying alternate join orders for building more cardinality models [3]. However, this could be expensive or infeasible depending on production settings. Therefore, we believe this is still an open problem.

Model variance. Learned models also need to be robust to changes in the workload. Although this goes against the notion of instance optimization, this is important in cloud systems due to the huge gap between the default and the optimal decisions, and the risk of good model falling off the cliff to very poor defaults. Thus, robustness is highly desirable property.

Coverage. Our micromodel approach allows to create fine grained and specialized models that are likely to be highly accurate. However, this also introduces the risk of not being able to cover the entire workload. Therefore, we see a trade-off between accuracy and coverage over the workload. Our recent work tried to employ an ensemble-based approach to strike a good middle ground [17]. However, striking the sweet spot between the two is still an open problem.

In-process scoring. The goal of Microlearner is to incorporate a large number of learned models into the optimization process. Typically, the SCOPE query optimizer would take up to a few seconds end to end for a reasonably sized query. However, with more complexity and more micromodels, the inference overheads need to be very low. The traditional approach is to serve inference from REST end points. However, with Microlearner, we want to score the micromodels in-process, e.g., using libraries such as ONNX [27]. Low inference time also dictates the featurization overhead and the choice of simple versus complex models. Inference latency is a challenge as the number of micromodels grow in Microlearner.

Tracking. Increasingly, there is a need to improve model tracking and governance. This is not just for audit purposed but also for service operators to be able to operations like locating the models currently in use, identifying the datasets they were trained on, invalidating models based on requirements such as GDPR, or simply just tracing their coverage and accuracy over time. For example, deployment of new SCOPE release should trigger re-training of the cardinality models since there might be new operators introduced or new optimizer rules added that need to be considered in cardinality models.

Troubleshooting. Finally, the service operators need to be

able to troubleshoot incidents. For example, for a customer support request, is there a way to disable a cardinality model? Disabling one cardinality model could impact many other jobs, so how we disable a set of models? What is the fallback strategy? Answering some of the questions are important for a seamless customer experience.

VI. Concluding Remarks

No one model fits all. Cloud workloads are highly heterogeneous and it is really hard to fit a single global model to the entire workload: it would require humongous training sets and very thorough featurization for learning such a model. In our experience, global models ended up being highly erroneous since they were unable to capture the large number of behaviors. Instead, learning smaller micromodels for specific patterns in the workload turns out to be more feasible. The downside is that micromodels may not cover the entire workload. One possibility is to combine multiple micromodels or combine them with a global model, similar to recursive-model index [21] or the mixture of experts approach [22]. Still, we believe there is an interesting interplay between accuracy and coverage that needs to be explored further.

How big is the black box. Given that the optimizer makes several key decisions, many of which are composed of each other, the question is what granularity to learn at. For instance, one could consider learning the output plan generated from an input query. Likewise, we could consider learning the cost model directly on the query without using cardinality as input, and so on. While bigger black boxes may seem more attractive since they can hide more complexity, they are also more difficult to get right. Learning fine grained micromodels on the other hand adds to the optimizer composability, understandability, and debuggability – all being huge pluses for system developers. Thus, there is a trade-off between hiding optimizer complexities and making it more amenable to developers.

Simple vs complex models. While popular machine learning tools provide sophisticated models and algorithms, much of the applied machine learning still relies on simpler models, e.g., linear models. This is because they are easier to featurize for, faster to train, and easier to debug or reason about. Thus, simple models are easier to deploy, though it comes at the cost of lower predictability.

Debugging / explainability. The optimizer codebase was more debuggable and understandable before, however, micromodels introduce black boxes into the system. This makes the lives of systems developers hard in the face of customer incidents or support requests. Therefore, being able to explain, debug, modify, and disable micromodels are going to be the software essentials before every release. This is partly because the system developers are not really machine learning experts, and partly because some models are inherently tough to explain.

System experiments. Finally, any software release involves an end-to-end validation using system experiments in a pre-production environment. Typically, these system experiments are designed by constructing a limited set of representative workload based on prior experience. Alternatively, domain experts add specific tests for new features based on their understanding of those features. Unfortunately, by introducing micromodels (for different features or specific models for each feature), system experiments become very hard because: (i) there is a large number of knobs to turn different micromodels on or off, (ii) these micromodels are hard to understand by domain experts and hence hard for them to test convincingly, and (iii) there are a limited set of resources in pre-production environment anyways. Thus, designing effective system experiments that can identify the set of production ready micromodels with high enough confidence within the constraints of the pre-production environments is a major challenge. We believe this would be interesting future work.

References

[1] T. Kraska *et al.*, "Sagedb: A learned database system," *CIDR*, 2019.
[2] R. Marcus *et al.*, "Neo: A Learned Query Optimizer," *PVLDB*, vol. 12, no. 11, pp. 48–57, 2019.
[3] C. Wu *et al.*, "Towards a Learning Optimizer for Shared Clouds," *PVLDB*, vol. 12, no. 3, pp. 210–222, 2018.
[4] I. Trummer *et al.*, "SkinnerDB: Regret-Bounded Query Evaluation via Reinforcement Learning," in *SIGMOD*, 2019, pp. 1153–1170.
[5] J. Ortiz *et al.*, "Learning State Representations for Query Optimization with Deep Reinforcement Learning," in *DEEM*, 2018, pp. 4:1–4:4.
[6] R. Chaiken *et al.*, "SCOPE: easy and efficient parallel processing of massive data sets," *PVLDB*, vol. 1, no. 2, pp. 1265–1276, 2008.
[7] J. Zhou *et al.*, "SCOPE: parallel databases meet MapReduce," *VLDB J.*, vol. 21, no. 5, pp. 611–636, 2012.
[8] "Azure Data Lake," https://azure.microsoft.com/en-us/solutions/data-lake.
[9] "AWS Athena," https://aws.amazon.com/athena/.
[10] "Google BigQuery," https://cloud.google.com/bigquery.
[11] A. Jindal *et al.*, "Peregrine: Workload optimization for cloud query engines," in *SoCC*, 2019.
[12] R. Ramakrishnan *et al.*, "Azure Data Lake Store: a hyperscale distributed file service for Big Data analytics," in *SIGMOD*, 2017, pp. 51–63.
[13] "Cloud Programming Simplified: A Berkeley View on Serverless Computing," https://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.pdf.
[14] C. Curino *et al.*, "Hydra: a federated resource manager for data-center scale analytics," in *NSDI*. USENIX, 2019.
[15] S. Qiao *et al.*, "Hyper Dimension Shuffle: Efficient Data Repartition at Petabyte Scale in SCOPE," in *VLDB*, 2019.
[16] L. Viswanathan *et al.*, "Query and Resource Optimization: Bridging the Gap," in *ICDE*, 2018, pp. 1384–1387.
[17] T. Siddiqui *et al.*, "Cost models for Big Data query processing: Learning, retrofitting, and our findings," in *SIGMOD*, 2020, pp. 99–113.
[18] R. Sen *et al.*, "AutoToken: Predicting peak parallelism for Big Data analytics at Microsoft," *PVLDB*, vol. 13, no. 12, pp. 3326–3339, 2020.
[19] A. Jindal *et al.*, "Computation Reuse in Analytics Job Service at Microsoft," in *SIGMOD*, 2018.
[20] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, Nov. 2011.
[21] T. Kraska *et al.*, "The case for learned index structures," in *SIGMOD*. ACM, 2018, pp. 489–504.
[22] N. Shazeer *et al.*, "Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer," *CoRR*, vol. abs/1701.06538, 2017.
[23] "Art. 17 GDPR. Right to erasure ('right to be forgotten')," https://gdpr.eu/article-17-right-to-be-forgotten/.
[24] "MLflow and Azure Machine Learning," https://docs.microsoft.com/en-us/azure/machine-learning/how-to-use-mlflow.
[25] S. Agarwal *et al.*, "Re-optimizing data-parallel computing," in *NSDI*. USENIX Association, 2012, pp. 21–21.
[26] E. Boutin *et al.*, "Apollo: scalable and coordinated scheduling for cloud-scale computing," in *OSDI*, 2014, pp. 285–300.
[27] "ONNX," https://onnx.ai/.