Towards a Learning Optimizer for Shared Clouds

Chenggang Wu[2], Alekh Jindal[1], Saeed Amizadeh[1], Hiren Patel[1], Wangchao Le[1]
Shi Qiao[1], Sriram Rao[3*]

[1]Microsoft
{aljindal,saamizad,hirenp,wanle,shqiao}@microsoft.com

[2]University of California, Berkeley
cgwu@berkeley.edu

[3]Facebook
sriramrao@fb.com

ABSTRACT

Query optimizers are notorious for inaccurate cost estimates, leading to poor performance. The root of the problem lies in inaccurate cardinality estimates, i.e., the size of intermediate (and final) results in a query plan. These estimates also determine the resources consumed in modern shared cloud infrastructures. In this paper, we present CARDLEARNER, a machine learning based approach to learn cardinality models from previous job executions and use them to predict the cardinalities in future jobs. The key intuition in our approach is that shared cloud workloads are often recurring and overlapping in nature, and so we could learn cardinality models for overlapping subgraph templates. We discuss various learning approaches and show how learning a large number of smaller models results in high accuracy and explainability. We further present an exploration technique to avoid learning bias by considering alternate join orders and learning cardinality models over them. We describe the feedback loop to apply the learned models back to future job executions. Finally, we show a detailed evaluation of our models (up to 5 orders of magnitude less error), query plans (60% applicability), performance (up to 100% faster, 3× fewer resources), and exploration (optimal in few 10s of executions).

PVLDB Reference Format:
Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, Sriram Rao. Towards a Learning Optimizer for Shared Clouds. *PVLDB*, 12(3): 210-222, 2018.
DOI: https://doi.org/10.14778/3291264.3291267

1. INTRODUCTION

> *The root of all evil, the Achilles Heel of query optimization, is the estimation of the size of intermediate results, known as cardinalities.* [27]

While it is well-known that poor cardinality estimation leads to inaccuracy in traditional query optimizers [17, 26, 18], the problem is even harder with big data systems. This is due to: (i) massive volumes of data which are very expensive to analyze and collect statistics on, (ii) presence of unstructured data that have schema imposed at runtime and hence cannot be analyzed a priori, and (iii) pervasive use of custom user code (e.g., UDFs) that embed arbitrary application logic resulting in arbitrary output cardinalities.

Cardinality estimates are even more important to shared cloud infrastructures, such as Google's BigQuery [6], Amazon's Athena [5], and Microsoft's Azure Data Lake [11]. These shared cloud infrastructures expose a *job service* abstraction for users to submit analytics queries written in a SQL-like query language [42, 9], and a *pay-as-you-use* model for users to pay only for the resources consumed per job (e.g., number of containers, size of containers, etc.). Cardinality estimates are crucial in determining the resources needed, and hence the monetary costs incurred, to execute a job in these shared infrastructures.

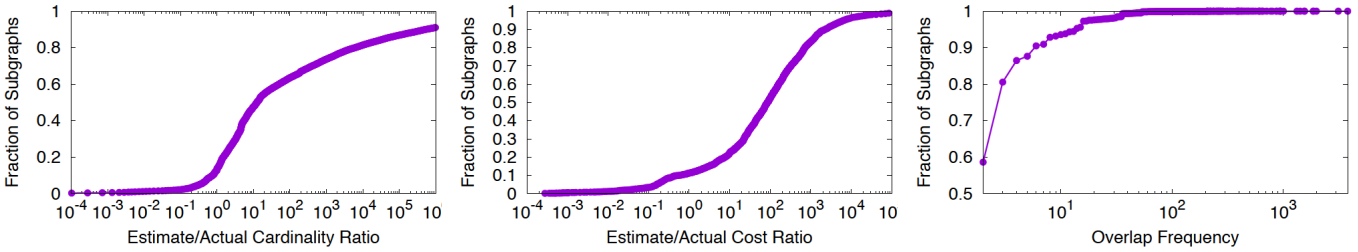
Prior techniques to improve cardinality estimates include the popularly used dynamic query re-optimization [19, 28, 8, 46], i.e., monitoring the actual cardinalities and adjusting the query plan as the query execution progresses. A major challenge of dynamic query re-optimization is determining when to re-optimize. On one hand, aggressive re-optimization can lead to increased overhead. On the other hand, a large portion of the query may have already been executed sub-optimally if re-optimization is performed conservatively. Moreover, adjusting the query plan in a distributed setting is expensive. Other feedback-based approaches either rely on adjustment factors [39] and are prone to error, or reuse the cardinalities of the exact same query subexpressions seen before [2] and have limited applicability.

In this paper, we explore a radically different approach to deriving intermediate cardinalities. Given that estimating cardinalities at compile-time is non-trivial for big data systems, we consider whether we can *learn* cardinalities from past executions. This is possible due to the popularity of shared cloud infrastructures [20] with large volumes of query workloads [34] that could be used for training. Furthermore, these shared cloud workloads are often recurring and overlapping in nature [22, 21, 20], providing significant opportunities to learn and apply a *cardinality model* multiple times.

We present CARDLEARNER, a machine learning based approach to improve cardinality estimates at each point in a query graph. Our key idea is to extract overlapping subgraph templates that appear over multiple query graphs and learn cardinality models over varying parameters and inputs to those subgraph templates. The resulting predictor is up to five orders of magnitude more accurate than the default cardinality estimators. This is an important step towards building a learning optimizer for modern shared cloud workloads. We further introduce an exploratory join ordering technique to explore alternate join orders (i.e., alternate subgraphs), build cardinality models over them, and eventually find the optimal join order. We have integrated CARDLEARNER with the SCOPE [9, 47] query optimizer and it will be available in an upcoming SCOPE release. In summary, our core contributions are as follows:

*Work done while at Microsoft.

Proceedings of the VLDB Endowment, Vol. 12, No. 3
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3291264.3291267



(a) Cardinality Accuracy (b) Cost Accuracy (c) Subgraph Overlaps

Figure 1: SCOPE workload from Cosmos clusters at Microsoft. Figures 1(a) and 1(b) compare the estimated/actual cardinalities and costs, while Figure 1(c) motivates the presence of overlaps in production workloads.

(1) We present CARDLEARNER, a learning approach to cardinality estimation in big data workloads. We motivate the problem of inaccurate cardinalities from production workloads at Microsoft. We further illustrate the overlapping nature of production workloads and how it facilitates learning cardinalities for subgraph templates. We then describe the key requirements and our design choices for building such a system in our production setting (Section 2).

(2) We walk through the journey of learning cardinality models and discuss the trade-offs in various approaches. Instead of learning one single giant model, we show how a large number of small models are helpful in achieving high accuracy as well as low overhead in terms of featurization (Section 3).

(3) We present an exploration technique to consider alternate join orders and avoid learning bias. We describe an exploratory join ordering algorithm that leverages prior executions to quickly prune the search space and consider only the promising join order candidates over a small number of executions (Section 4).

(4) We describe our feedback loop to inject the predicted cardinalities back to the optimizer. This includes offline workload analysis, parallel training, and annotations for future queries (Section 5).

(5) Finally, we show a detailed experimental evaluation, including model evaluation, plan evaluation, performance evaluation, and exploration evaluation, over production workloads at Microsoft. Our results show that the learning-based approach is applicable to 60% of the jobs and 50% of the subgraphs, has 75^{th} percentile cross-validation error of 1.5%, which is five orders of magnitude lower than the default optimizer error, and results in plans which have lower latency (25% less), processing time (55% less), as well as containers used (60% less). The exploration technique is further able to find the optimal join orders in few $10s$ of executions for fairly sized schemas (Section 6).

2. CARDLEARNER OVERVIEW

Motivation. In this section, we illustrate the problem of cardinality and cost estimation in SCOPE [9, 47] workloads at Microsoft, which motivates our approach. SCOPE data processing system processes multiple exabytes of data over hundreds of thousands of jobs running on hundreds of thousands of machines. These jobs are written in a SQL-like language and authored by thousands of internal developers, across different business units in Microsoft, to draw insights from the usage of various Microsoft products. To evaluate the cardinality and cost estimates in SCOPE, we analyzed one day's workload, consisting of tens of thousands of jobs from thousands of internal users, from one of the largest SCOPE customers, the Asimov system [4]. The Asimov system uses SCOPE to analyze telemetry data from millions of Windows devices in a shared cloud infrastructure, i.e., thousands of users processing shared datasets using a shared set of compute resources.

Figure 1(a) shows the cumulative distribution of the ratio of estimated and actual cardinalities of different subgraphs. Our results show that a very tiny fraction of the subgraphs have estimated cardinalities matching the actual ones, i.e., the ratio is 1. Almost 15% of the subgraphs underestimate (up to 10,000x) and almost 85% of the subgraphs overestimate (up to 1 million times!). Figure 1(b) shows the cumulative distribution of the ratio of estimated and actual processing costs of different subgraphs. We see that the estimated costs are as much off as the cardinality — up to 100,000x overestimated and up to 10,000x underestimated than the actual costs. This is because the cost models are built considering cardinality as the key input and therefore estimated costs are directly correlated with the estimated cardinality. Hence, cardinality is indeed the root of the problem.

We also computed the Pearson correlation coefficient between the estimated and actual cardinalities/costs, and they both turned out to be very low. As discussed before, estimating cardinalities in a big data system like SCOPE is challenging for several reasons, including unstructured data and user defined operators (UDOs). In its current state, the SCOPE query processing system derives reference selectivities, for different operator classes, from the most important workloads. Obviously, these selectivities are not applicable to all scenarios, offering a huge potential for improvement. Also, making simple adjustments to these selectivities do not help, as we discuss in detail in Section 3.1.

SCOPE workloads are also overlapping in nature: multiple jobs access the same datasets in the shared cloud and end up having common subgraphs across them. These jobs are further recurring in nature, i.e., they are submitted periodically with different parameters and inputs. Figure 1(c) shows the cumulative distribution of subgraph overlap frequency. We see that 40% of the subgraphs appear at least twice and 10% appear more than 10 times. Thus, we could leverage subgraph overlaps to learn cardinality models. By improving cardinality estimates, our goal is to improve plan quality, i.e., produce plans with lower costs, and to reduce the resource consumption caused by overestimation, i.e., avoid over-provisioning a large number of containers, each processing a tiny dataset.

Note that while we focus on big data in shared clouds due to the presence of overlaps in them, the problem of cardinality estimation is also relevant for other clouds, short running jobs, or even traditional databases. It is equally possible for these other environments to have overlapping workloads and hence the learning opportunities. Exploring these will be a part of future work. Likewise, the over-provisioning problem would be applicable to all container-based data processing environments. Finally, the cost associated with improving cardinality estimation is a system cost to improve system efficiency and attract more customers, even though each of them might be paying less in the pay-as-you-go model.

Requirements. Our key requirements derived from the current production setting are as follows:

R1. Being minimally invasive, i.e., we do not want to completely rewrite the existing optimizer. Furthermore, the optimizer should have the flexibility to decide whether to use the improved cardinalities, and be able to overwrite them with any user-provided hints.

R2. Having an offline feedback loop to learn the cardinality models outside of the critical path and use them multiple times before updating with new ones. This is because workload traces on our clusters are collected and post-processed offline.

R3. Having low compile time overhead to the existing optimizer latencies (typically 10s to 100s of milliseconds), i.e., cardinality improvement mechanisms should be lightweight.

R4. Finally, the improved cardinalities should be explainable, i.e., we do not want to add a black box which is difficult to reason about. This is important in SCOPE like cloud services [6, 5], where customer concerns need to be resolved quickly.

Approach. We apply a learning-based approach to improve cardinalities using past observations. Instead of using state-of-the-art operator based learning [39] (discussed in Section 3.1), we consider subgraphs and learn their output cardinalities. Rather than learning a single giant model to predict cardinalities for all possible subgraphs, we learn a large number of smaller models (few features), one for each subgraph template in the workload. These models are highly accurate as well as much easier to understand (**R4**). Smaller feature set also makes it easier to featurize during prediction, adding minimal compile time overhead (**R3**). Furthermore, since we learn over subgraph templates, we can train our models periodically over new batches of data and outside of the critical path of query processing (**R2**). Finally, we provide the improved cardinalities as annotation hints to the query that could later be applied wherever applicable by the optimizer, i.e., we do not overwrite the entire cardinality estimation mechanism and the optimizer can still choose which hints to apply (**R1**).

In the rest of the paper, we describe our approach to learning cardinality models in Section 3, introduce algorithms to avoid learning bias in Section 4, discuss system details of our feedback loop in Section 5, and present a detailed evaluation in Section 6. We discuss related work in Section 7. Finally, Section 8 concludes.

3. LEARNING CARDINALITY MODELS

Traditional query optimizers incorporate a number of heuristics to estimate the cardinalities for each candidate query subgraph. Unfortunately, these heuristics often produce inaccurate estimates, leading to significantly poor performance [17, 26, 39]. Given large prior workloads in production systems, the question is whether we can *learn* models to predict the cardinalities. The rest of this section describes the various aspects we considered towards answering this question.

3.1 Adjustment Factors

Before considering sophisticated learning models, one important question is whether we can simply apply adjustment factors (i.e., the ratio between the actual selectivity and the estimated selectivity of a given operator) to cardinality estimates and get satisfactory results. This is similar to the approach followed in LEO [39]. The adjustment factor approach can be viewed as a very simple linear model, with its slope being the adjustment factor and its only feature being the estimated selectivity. This approach suffers from two major problems: First, adjusting selectivity using a constant factor assumes that the relationship between the input and output cardinality is linear. However, this assumption rarely holds given the presence of complex operators such as join, aggregation, and UDO in production workloads. Therefore, a single adjustment factor is not enough to model the relationship between the input and output cardinality. Second, this approach considers estimated selectivity as the only relevant feature to adjust the output cardinality. This is limiting because there are many other factors that contribute to the output cardinality, especially under the presence of parameterized user code. To illustrate, Table 1 shows the input and output cardinalities from multiple instances of a recurring user-defined reducer from the SCOPE workload (see Section 2).

Table 1: Cardinalities of a recurring user-defined reducer.

Reducer Instance	Input Cardinality	Output Cardinality
R_1	672331	1596
R_2	672331	326461
R_3	672331	312
R_4	672331	2
R_5	672331	78
R_6	672331	1272
R_7	672331	45
R_8	672331	6482

Although the input cardinality to the reducer is the same across all instances, the output cardinalities are very different. This is because the output cardinality depends on the parameters of the reducer, which are different across instances. Hence, simple adjustment factors will not work for complex queries with such user code. To validate that, we computed the percentage error and Pearson correlation (between the predicted and actual cardinality) over this dataset, which yields over 1 million percent error and Pearson correlation of 0.38. Therefore, the adjustment factor approach does not help in this situation.

We extended the above analysis to all subgraphs in the SCOPE workload (see Section 2) that have the same logical expression[1]. The average coefficient of variation of output cardinalities turns out to be 22%, 75^{th} percentile being 3.2% and 90^{th} percentile being 93%. Thus, cardinalities vary significantly and simple adjustment factors will not work. If we use the average adjustment factor for each subgraph, the 75^{th} and 90^{th} percentile errors are 78% and 840% respectively, compared to 1.5% and 32% for the learned models (See Section 6.1.2).

Given the above limitations, we consider using more sophisticated learning techniques. We introduce the granularity of our learning models in Section 3.2, and discuss how we enhance the feature set and pick the model in Sections 3.3 and 3.4 respectively.

3.2 Granularity of Learning

Our goal is to obtain accurate output row counts for every subgraph in each of the queries. These subgraphs are rooted at one of the internal operator nodes and cover all prior operators from that node. We consider different learning granularities for a subgraph, depending on what part of the subgraph is considered fixed, as shown in Table 2 below.

Table 2: Possible granularities for learning cardinality.

Subgraph Type	Logical Expression	Parameter Values	Data Inputs
Most Strict	✓	✓	✓
Template	✓	✗	✗
Most General	✗	✗	✗

The top row in Table 2 shows one extreme, the most strict subgraphs, where the logical expression, parameter values, and data inputs are all fixed. In this case, we simply record the subgraph cardinalities observed in the past and reuse them in future occurrences of the exact same subgraphs. This is similar to the approach followed in ROPE [2]. While these feedbacks are the most accurate, such strict matches are likely to constitute a much smaller

[1]Logical expression includes the predicates and the schemas of input datasets but excludes the parameter values and the content of the input datasets.

fraction of the total subgraphs in the workload (e.g., less than 10% on our production workloads). Hence, low applicability is the main challenge with the most strict subgraph matches.

The bottom row in Table 2 shows the other extreme, where none of the logical expressions, parameter values, and data inputs are fixed. In this case, we essentially learn a single global model that can predict cardinalities for all possible subgraphs, i.e., having full applicability. However, it turns out that building a single global model is impractical for a number of reasons: (i) *feature engineering*: featurizing the logical expression of the subgraph is challenging since it is difficult to express a variable-size subgraph as a fixed-size feature vector without losing the structure of the graph, which we assume have strong predictive signal. (ii) *large-scale training*: a very large set of training data is required to train one single model, which in turn needs powerful scale-out machine learning tools to be developed. (iii) *prediction latency*: the single model approach requires a large number of features, and extracting all features during the query compilation phase is *at odds* with the pressing needs of low compile time; in particular, getting features that relate to input data distribution (such as max, min, number of distinct values) could require preprocessing that is simply not possible for ad-hoc queries.

We take a middle ground. Our approach is to learn cardinalities for each *template* subgraph (Table 2), with varying parameters and inputs. This has a number of advantages:

(1) *Easier to featurize:* We no longer need to featurize the logical expression. Furthermore, since subgraphs with the same logical expression have the same input data schema, the data distributions remain roughly the same and the only data feature that typically matters is the input size, which could be easily retrieved from the database catalog. Furthermore, the parameter values provided to the logical expression could be readily used as features. Therefore, learning at this granularity makes featurization much simpler.

(2) *Higher applicability:* Compared to the most strict subgraphs, learning cardinalities for templates gives higher applicability as it allows both parameters and data inputs to vary (Table 2).

(3) *Higher accuracy:* It is challenging to have high accuracy in a single model, due to the non-linear nature of the target cardinality function. Instead, a large number of smaller models allows us to capture the non-linearity of each subgraph more accurately.

(4) *Offline feedback loop:* Since subgraph templates allow for inputs and parameters to vary, we could periodically train the model and use it to predict future subgraphs that have matching logical expressions. Moreover, since training happens offline, its overhead does not lie on the critical path of query processing. We discuss in Section 6 how to detect when the models become fairly inaccurate and retrain them.

3.3 Feature Engineering

Below we discuss the features we use to train our models and analyze the impact of each feature on different subgraph models.

3.3.1 Feature Selection

There are three types of features that we consider, listed together in Table 3. First, we extract metadata such as the name of the job the subgraph belongs to and the name of the input datasets. These metadata attributes are important as they could be used as inputs to user defined operators. In fact, the reason that leads to the orders-of-magnitude difference in the output cardinality between the first and the second row in Table 1 is due to the difference in the name of the job (everything else is the same for these two observations).

Second, we extract the input cardinalities of all input datasets. Intuitively, the input cardinality plays a central role in predicting the output cardinality (similar to LEO's assumption [39]). In order to account for operators (joins, aggregations, and UDO) that can lead to a non-linear relationship between the input and output cardinality, we also compute the squared, squared root, and the logarithm of the input cardinality as features.

Table 3: The features used for learning cardinality.

Name	Description
JobName	Name of the job containing the subgraph
NormJobName	Normalize job name
InputCardinality	Total cardinality of all inputs to the subgraph
$Pow(\text{InputCardinality}, 2)$	Square of InputCardinality
$Sqrt(\text{InputCardinality})$	Square root of InputCardinality
$Log(\text{InputCardinality})$	Log of InputCardinality
AvgRowLength	Average output row length
InputDataset	Name of all input datasets to the subgraph
Parameters	One or more parameters in the subgraph

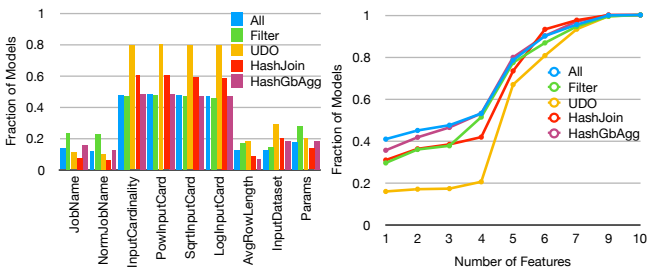
Finally, since the parameters associated with operators, such as filters and UDOs, can have a big impact in the output cardinality, we also extract these parameters as features.

3.3.2 Feature Analysis

We now analyze the features that contribute to our model's prediction. We analyze the Poission regression models under the production training data from one of the largest customers in SCOPE (described earlier in Section 2). We consider Poisson regression since it offers the best performance, as shown later in Section 6. For each model, the features that do not contribute to the prediction are given zero weights. Hence, inspecting the features with non-zero weights gives us insight about what features contribute more to the prediction (important for R4 from Section 2). In Figure 2(a), for each feature (x-axis), we compute the fraction of the models where the feature has non-zero weights (y-axis). Since each model can have different number of parameters as features, we group these parameters into one feature category named 'Parameters' for ease of presentation. Across all the models that we trained, we notice that InputCardinality plays a central role in model prediction as it has non-zero weights in near 50% of the models. It is worth noting that the squared, squared root, and logarithm of the input cardinality also have big impact on the prediction. In fact, the fractions are a bit higher than InputCardinality. Interestingly, all other features also have noticeable contribution. Even the least significant feature, AvgRowLength, has non-zero weights in more than 10% of the models.

We further group the models based on the root operator of the subgraph template, and analyze models whose root operators are Filter, UDO, Join, and Aggregation. For Join and UDO, we notice that the importance of cardinality and input dataset features go up significantly, possibly due to complex interaction between different datasets for Joins and ad-hoc user-defined data transformations for UDOs. For Filter, it is not surprising to see that Parameters contribute a lot more, e.g., the parameter values to the filter operator can have big impact on the output cardinality. For Aggregation, we see the AvgRowLength matters a lot less because a large fraction of the aggregation queries produce a single number as output, which has the same row length. To summarize, Figure 2(a) shows that a lot of features other than InputCardinality contribute to cardinality prediction, and models with different operators have different set of important features.

Figure 2(b) shows the cumulative distribution of the number of non-zero weight features our models contain. Overall, we see that more than 55% of the models have at least 3 features that contribute to the prediction, and 20% of the models have at least 6 features. It is worth noting that for models whose root operators are UDOs, more than 80% of the models have at least 5 contributing features. This confirms that subgraphs with complex operators need a num-



(a) Feature Contribution (b) Feature CDF

Figure 2: Fraction of models that contain each feature with non-zero weight (2(a)), and the cumulative distribution of the number of non-zero weight features a model contains (2(b)).

ber of features to accurately predict the output cardinality, as opposed to relying solely on the input cardinality (Section 3.1).

3.4 Choice of Model

We experimented with three different types of machine learning models: linear regression (LR) [31], Poisson regression (PR) [13], and multi-layer perceptron (MLP) neural network [14]. While LR is a purely linear model, PR is slightly more complex and is considered a generalized linear model (GLM) [30]. MLP, on the other hand, provides us with a fully non-linear and arbitrarily complex predictive function.

The main advantage of using linear and GLM models is their *interpretability*, which is crucial for our requirement R4 from Section 2. In particular, it is easy to extract the learned weights associated with each feature so that we can explain which features contribute more to the output cardinality in a postmortum analysis. This can be useful for many practical reasons as it gives analysts an insight into how different input query plans produce different output cardinalities. The simplicity and explainability can however come at a cost: the linear model may not be sufficiently complex to capture the target cardinality function. This is referred to as the problem of *underfitting* which puts a cap on the accuracy of the final model regardless of how big our training data is. Also, note that LR can produce negative predictions which are not allowed in our problem since cardinalities are always non-negative. Though, we can rectify this problem by adjusting the model output after-fact so that it will not produce negative values. PR, on the other hand, does not suffer from this problem as by definition it has been built to model (non-negative) *count-based* data.

On the other extreme, MLP provides us with a much more sophisticated and richer modeling framework that in theory is capable of learning the target cardinality function regardless of its complexity, given that we have access to sufficient training data. In practice, however, training and using a MLP for cardinality estimation is way more challenging than that of LR or PR for some fundamental reasons. First, as opposed to LR and PR, using a MLP requires careful designing of the neural network architecture as well as a significant hyper-parameter tuning effort. Secondly, if we do not have enough training data for a given subgraph template, depending on its complexity, it is very easy for a MLP to just *memorize* the training examples without actually learning how to *generalize* to future examples, also known as the *overfitting* problem in machine learning. Finally, it is quite difficult to explain and justify the output of MLP for human analysts even though the output might as well be quite an accurate prediction of the cardinality.

To illustrate the effectiveness of different models, we compare the percentage error and Pearson correlation (between the actual and predicted cardinality) across different learning approaches over the workload used in Section 3.1 (multiple instances of a reducer

Table 4: Comparing adjustment factor with other approaches.

Model	Percentage Error	Pearson Correlation
Default Optimizer	2198654	0.41
Adjustment Factor (LEO)	1477881	0.38
Linear Regression	11552	0.99
Neural Network	9275	0.96
Poisson Regression	696	0.98

that takes different parameters). Table 4 shows that although LEO's approach improves slightly over the default optimizer estimates, it still has very high estimation error and low correlation. On the other hand, our learning models (linear regression, poisson regression and neural networks) can reduce the error by orders-of-magnitude and provide very high correlation. Note that although significantly better than the adjustment factor approach, our models still generate high prediction errors (ranging from 696% to 11552%). The reason is that for this workload, input cardinality does not contribute to the variation of output cardinality, as it is fixed across all reducer instances. Moreover, most of the parameter inputs to the reducer take non-numerical values that are difficult to map to output cardinality. These make it difficult to predict the exact value of the output cardinality. However, the high Pearson correlation indicates that there is a near-linear relationship between the predicted cardinality (cost) and the actual cardinality (cost). This implies that the query plan picked by the optimizer is highly likely to be optimal, regardless of the exact value of the predicted costs.

3.5 Limitations

Although our feature-based learning could achieve very good performance, as shown in Section 6, there are a few limitations to this approach.

First, our feature-based framework cannot make predictions for unobserved subgraph templates. Of course, we can collect more data, i.e., observing more templates, during the training phase, but the model still cannot improve the performance of ad-hoc queries with new subgraph templates.

Second, since we train a model for each subgraph template, the number of models grows linearly with the number of distinct subgraph templates in the workload. Therefore, in case of limited storage budget, we need to rank and filter the models based on their effectiveness in fixing the inaccurate cardinality estimates.

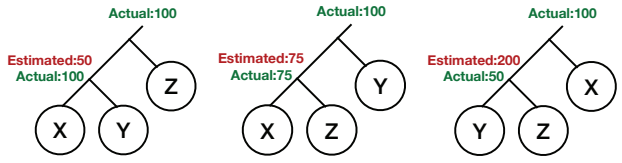
Finally, recall that a query optimizer chooses an execution path with the lowest cost. However, it may end up comparing the cost between two paths, where the cost of the first path is computed using the learned model's predicted cardinality and the cost of the second path is computed using the optimizer's default cardinality estimation (due to missing subgraph template models). Comparing these two costs could lead to inaccuracy as the optimizer's default cardinality estimation could be overestimating or underestimating the cardinality heavily.

The first limitation is beyond the scope of this paper. We discuss solutions to the second limitation in Section 6.1.1 and describe our approach to address the third limitation in Section 4.

4. AVOIDING LEARNING BIAS

Learning cardinalities can improve the cost estimates of all previously executed subgraph templates. However, since only one plan is executed for a given query, we still have inaccurately estimated costs for other subgraphs in that query. For example, Figure 3 shows three alternate plans for joining relations X, Y, and Z.

Since $X \bowtie Y$ (plan 1) has lower estimated cost than $X \bowtie Z$ and $Y \bowtie Z$, it will be picked (for illustration purpose we assume the cost model here only considers the output cardinality of the subgraph). Once plan 1 is executed, we know that the actual cardinality of $X \bowtie Y$ is 100, which is higher than the estimated cardinality of $X \bowtie Z$. Therefore, if we execute the same query again, the



(a) Plan 1 (b) Plan 2 (c) Plan 3

Figure 3: Candidate exploration plans with estimated and actual cardinalities for $X \bowtie Y \bowtie Z$.

Algorithm 1: EarlyPruning

Input : Relation r, Plan p, Query q, RuntimeCosts c, CardModels m
Output: Return null if pruning is possible; otherwise the input plan with updated cardinality, if possible.

```
1  if m.Contains(r) then
2      p.UpdateCard(m.Predict(r))
3  if c.Contains(p) & c.GetBestCost(q) < c.GetCost(p) then
       // prune as outer is more expensive than an
          overall query plan
4      return null
5  return p
```

optimizer will pick plan 2. However, even though $Y \bowtie Z$ is the cheapest option, it is never explored since the estimated cardinality of $Y \bowtie Z$ is higher than any of the actual cardinalities observed so far. Thus, we need a mechanism to explore alternate join orders and learn cardinality models of those alternate subgraphs, which might have higher estimated costs but turn out to be cheaper.

4.1 Exploratory Join Ordering

We now present an exploratory join ordering technique to consider alternate join orders, based on prior observations, and ultimately discover the best one. The core idea is to leverage existing cardinality models and actual runtime costs of all previously executed subgraphs to: (i) quickly explore alternate join orders and build cardinality models over the corresponding new subgraphs, and (ii) prune expensive join paths early so as to reduce the search space. Having cardinality models over all possible alternate subgraphs naturally leads to finding the best join order. We present our key components below.

Early pruning. The number of join orders are typically exponential and executing all of them one by one is simply not possible, even for a small set of relations. Therefore, we need to quickly prune the search space to only execute the interesting join orders. Our intuition is that whenever a subgraph plan turns out to be more expensive than a full query plan, we could stop exploring join orders which involve that subgraph plan. For instance, if $A \bowtie C$ is more expensive than $((A \bowtie B) \bowtie C) \bowtie D$, then we can prune join orders $((A \bowtie C) \bowtie B) \bowtie D$ and $((A \bowtie C) \bowtie D) \bowtie B$, i.e., all combinations involving $A \bowtie C$ are discarded as the total cost is going to be even higher anyways.

Algorithm 1 shows the pseudocode for early pruning described above. The input relations correspond to the subgraph to be evaluated for pruning. The algorithm first updates the cardinality of the subgraph using the predicted value if a model is available (line 1-2). It then checks the RuntimeCosts cache to see whether the cost of the candidate plan is more expensive than the cost of the best query plan[2]. If so, the plan is pruned; otherwise it is kept.

Exploration comparator. The goal of exploratory join ordering is to learn cardinality models for alternate subgraph templates. Thus, for two equivalent plans, we want to pick the one which maximizes the number of new subgraph templates observed. This is in contrast to the typical approach of picking the cheapest plan amongst equivalent query plans. Figure 4 illustrates the idea, where the planner first executes the plan shown in Figure 4(a) and then considers next plan choices 1 and 2 for other runs of this recurring query, shown in Figure 4(b) and 4(c). The planner makes only one new observation with plan 1, namely $A \bowtie B \bowtie D$, as $A \bowtie B$ and $A \bowtie B \bowtie D \bowtie C$ (which is equivalent to $A \bowtie B \bowtie C \bowtie D$) have already been observed. However, with plan 2, the planner makes two new observations, namely $C \bowtie D$ and $C \bowtie D \bowtie A$. Thus, plan 2 is better in terms of the number of new observations. Alternatively, in case $C \bowtie D$ had appeared in some other queries, plans 1 and 2 would have had the same number of observations.

[2]It could be the observed cost from the exact same subgraph/query or the predicted cost computed using a learned cardinality model from the recurring subgraph/query.

Algorithm 2: ExplorationComparator

Input : Plan p1, Plan p2, Ranking ranking, RuntimeCosts c
Output: Return true if p1 is better than p2; false otherwise.

```
1  h1 = NewObservations(c, p1)
2  h2 = NewObservations(c, p2)
3  begin
4      switch ranking do
5          case OPT_COST
6              return (p1.cost < p2.cost)
7          case OPT_OBSERVATIONS
8              return (h1 > h2) | (h1==h2 & p1.cost < p2.cost)
9          case OPT_OVERHEAD
10             return (p1.cost << p2.cost) | (p1.cost ≈ p2.cost & h1 > h2)
```

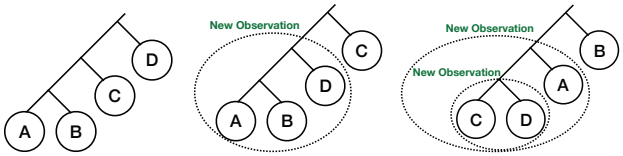
Algorithm 3: ExploratoryPlanner

Input : Query q, Ranking ranking, RuntimeCosts c, CardModels m
Output: Left-deep plan for the query q.

```
1  Relation [] rels = LeafRels(q)                  // relations to join
2  Map <Relation,Plan > optPlans = {}
3  foreach r ∈ rels do
4      optPlans[r] = ScanPlan(r)                   // generate scan plans
   // perform left-deep bottom-up enumeration
5  foreach d ∈ [1, |R| − 1] do
6      foreach outer : outer ⊂ R, |outer| = d do
7          foreach inner : inner ∈ (R − outer) do
8              Plan pOuter = optPlans[outer]
9              Plan pInner = optPlans[inner]
10             pOuter = EarlyPruning(outer, pOuter, q, c, m)
11             pInner = EarlyPruning(inner, pInner, q, c, m)
12             if pOuter==null || pInner==null then
13                 Continue
14             Plan p = OptJoin(pOuter, pInner)        // join algo
15             Plan pOpt = optPlans[p.rel]
16             if (pOpt==null) || ExplorationComparator(p, pOpt,
               ranking, c) then
17                 optPlans[p.rel] = p                 // add plan
18 return optPlans[q];
```

Algorithm 2 shows the exploration plan comparator. In addition to the standard comparison to minimize cost (Lines 5–6), we provide a mode to maximize the number of new observations (Line 7). In case of a tie, we pick the plan with the lower cost in order to keep the execution costs low (Line 8). To reduce the overhead even further, we could pick the plan with higher number of new observations only if both plans have similar cost (Lines 9–10). The exploration comparator provides knobs to the planner to explore alternate join orders over recurring queries.

Exploratory planner. We now describe how the early pruning strategy and the exploration plan comparator can be integrated into a query planner for exploratory join ordering. Algorithm 3 shows the exploratory version of System R style bottom-up query planner,



(a) Initial Plan (b) Next Plan 1 (c) Next Plan 2

Figure 4: Candidate exploration plans for a recurring query $A \bowtie B \bowtie C \bowtie D$.

also sometimes referred to as the Selinger planner [36]. The planner starts with leaf level plans, i.e., scans over each relation, and incrementally builds plans for two, three, and more relations. For each candidate outer and inner plans, we check if we can prune the search space (Lines 10–13). Otherwise, we compare (using the exploration comparator) the current plan to join outer and inner with the previous best seen before (Lines 14–16), and update the best plan if it turns out to be better (Line 17). Finally, we return the best plan for the overall query (Line 18).

We could likewise extend other query planners, e.g., top-down [16] or randomized [44] query planners, to explore alternate join orders and eventually find the best one. Essentially, we have a three-step process to convert a given query planner into an exploratory one: (i) Iterate over candidate plans using the planner's enumeration strategy, e.g., bottom-up, top-down, etc., (ii) Add a pruning step in the planner to discard subgraphs based on the costs of prior executions, i.e., subgraphs that were more expensive than the full query need not be explored anymore. This is in addition to any existing pruning in the planner, and (iii) Consider the number of new observations made when comparing and ranking equivalent plans. We could also incorporate costs by breaking ties using cheaper plans, or by considering observations only for plans with similar costs.

Planner overhead. Now we analyze the overhead of *ExploratoryPlanner*. Note that only three new lines (10, 11, 16) are added to the inner loop of the Selinger planner. The first two lines (10, 11) are invocation to *EarlyPruning*, whose complexity is $O(1)$ (it consists of hash table lookups and model evaluation, both of which are performed in constant time). Line 16 invokes *ExplorationComparator*, which consists of cost lookups and comparisons ($O(1)$). Note that determining the number of newly observed subgraph templates (line 1-2 of Algorithm 2) seems to require traversing through the entire candidate plan. In practice, however, this number is being updated incrementally and stored in the *optPlans* hash table, and therefore only incurs $O(1)$ overhead (e.g. for plan $(A \bowtie B) \bowtie C$, we only need to know if subgraph template ABC is new because we already have the new observation count for plan $A \bowtie B$ from prior iterations.). In summary, *ExploratoryPlanner* adds negligible complexity to the Selinger planner, and is consistent with our low compile time overhead requirement (R3) from Section 2.

4.2 Execution Strategies

We now discuss different execution strategies for exploratory join ordering with CARDLEARNER.

Leveraging recurring/overlapping jobs. Given the recurring and overlapping nature of production workloads at Microsoft, as described earlier in Section 2, the natural strategy is to run multiple instances of jobs (or overlaps) differently, i.e., apply the exploratory join ordering algorithm and get different plans for those instances. We could further *eagerly* run every instance of every job differently until we have explored all alternative subgraphs, i.e., we have cardinality models for those alternate subgraphs and can pick the optimal join orders. Or, we could *lazily* run every other instance of every other job differently to limit the number of expensive runs.

Static workload tuning. Instead of the above pay-as-you-go model for learning cardinality models for alternate subgraphs, we can tune queries upfront by running multiple trials, each with different join ordering, over the same static data. We show in Section 6.4 how our proposed techniques could quickly prune down the search space, thereby making the number of trials feasible even for fairly complex jobs.

Reducing exploration overhead. Since the actual costs of unseen subgraphs are unknown, exploratory join ordering could introduce significant runtime overheads. A typical technique to mitigate this is to perform pilot runs over a sample of the data [24]. Similar sample runs have been proposed for resource optimization [33], i.e., for finding the best hardware resources for a given execution plan. We could likewise use samples to learn cardinality models during static workload tuning. Samples could be built using the traditional apriori sampling [3], or the more recent just-in-time sampling [23].

5. THE FEEDBACK LOOP

In this section, we describe the end-to-end feedback loop to learn cardinality models and generate predictions during query processing. Figure 5 shows the architecture. Below we walk through each of the components in detail.

5.1 Workload Analyzer

The first step in the feedback loop is to collect traces of past query runs from different components, namely the compiler, optimizer, scheduler, and runtime. The SCOPE infrastructure is already instrumented to collect such traces. These traces are then fed to a workload analyzer, which (i) reconciles the compile-time and runtime statistics, and (ii) extracts the training data, i.e., all subgraphs and their actual cardinalities. Combining compile-time and runtime statistics requires mapping the logical operator tree to the data flow that is finally executed. To extract subgraphs, we traverse the logical operator tree in a bottom-up fashion and emit a subgraph for every operator node. For each subgraph, we extract the parameters by parsing the scalar operators in the subgraph, and extract the leaf level inputs by tracking the operator lineage. We use a unique hash, similar to plan signatures or fingerprints in earlier works, that is recursively computed at each node in the operator tree to identify the subgraph templates. Note that the leaf level inputs and the parameter values are excluded from the computation as subgraphs that differ only in these attributes belong to the same subgraph template. Finally, for each subgraph, we extract the features discussed in Section 3 and together with the subgraph template hash send them to the parallel trainer described below.

5.2 Parallel Trainer

Given that we have a large number of subgraph templates, and the model trained from one subgraph template is independent of others, we implement a parallel model trainer that can significantly speed up the training process. In particular, we use SCOPE to partition the training data for each subgraph template, and build the cardinality model for each of them in parallel using a reducer. Within each reducer, we use Microsoft's internal machine learning library to train the model. This library supports a wide range of models, including the ones discussed in Section 3, that can be used to predict cardinality. In addition to the model, the reducer also emits the training error and the prediction error for the ten-fold cross validation [40]. The reducer can also be configured to group these statistics by the type of the root operator of the subgraph template. This can help us investigate which type of subgraph template model is more effective compared to the optimizer's default estimation. The trained models are stored in a model server described below.

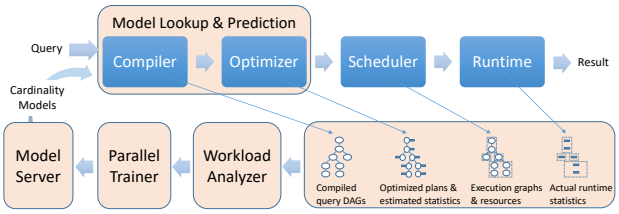


Figure 5: The feedback loop architecture.

5.3 Model Server

The model server is responsible for storing all the models trained by the parallel trainer. For each subgraph template hash, the server keeps track of the model along with its accuracy improvement (measured in the ten-fold cross validation) over the optimizer's default estimation. Models with high accuracy improvement are cached into the database to improve the efficiency of model lookup. Note that caching all models into the database is impractical due to limited storage resources. Since SCOPE job graphs can have hundreds of nodes and hence cardinality models, the model server builds an inverted index on the job metadata (which often remains the same across multiple instances of a recurring job) to return all relevant cardinality models for a given job in a single call.

5.4 Model Lookup & Prediction

The compiler and optimizer are responsible for model lookup and prediction, as shown in Figure 5. First, the compiler fetches all relevant cardinality models for the current job and passes them as annotations to the optimizer. Each annotation contains the subgraph template hash, the model, and the accuracy improvement. Thereafter, the optimizer prunes out the false positives by matching the subgraph template hash of the model with the hashes of each subgraph in the job graph. For matching subgraphs, the optimizer generates the features and applies them to the corresponding model to get the predicted cardinality. Either of the compiler or the optimizer could prune models with sufficiently low accuracy improvement. In addition, any row count hints from the user (in their job scripts) still supersede the predicted cardinality values, in accordance to requirement R1 from Section 2.

5.5 Retraining

We need to retrain the cardinality models for two reasons: (i) applying cardinality predictions would result in new query plans and new subgraph templates, and hence we need to retrain until the plans stabilize, and (ii) the workloads change over time and hence many of the models are no longer applicable. Therefore, we need to perform periodic retraining of the cardinality models to update existing models as well as adding new ones. We do this by keeping track of the cardinality models' *applicability*, i.e., the fraction of the subgraphs and jobs for which the models are available, and retrain when those fractions fall below a threshold. We show in our experiments (Section 6.1.3) that one month is a reasonable time to retrain our models.

5.6 Exploration

Exploratory join ordering executes alternate subgraphs that could be potentially expensive. Due to the SLA sensitivity of our production workloads, we need to involve humans (users, admins) in the loop in order to manage the expectations properly. Therefore, our current implementation runs the exploratory join ordering algorithm separately to produce the next join order given the subgraphs seen so far. Users can then enforce the suggested join order using the `FORCE ORDER` hint in their job scripts, which is later enforced by the SCOPE engine during optimization. Users can apply these hints to their recurring/overlapping jobs, static tuning jobs, or pilot runs over sample data. Note that involving human-in-the-loop is not fundamental to our approach and we only do that for exploration to manage user expectations. Alternatively, we could run the exploration phase on a sample dataset or over a static offline workload, as discussed in Section 4.2.

6. EXPERIMENTS

We now present an experimental evaluation of CARDLEARNER over the same dataset as discussed in Section 2, i.e., one days worth of jobs consisting of tens of thousands of jobs from one of the largest customers on SCOPE, the Asimov system [4]. The goals of our evaluation are four-fold: (i) to evaluate the quality of the learned cardinality models, (ii) to evaluate the impact of feedback loop on the query plans, (iii) to evaluate the improvements in performance, and (iv) to evaluate the effectiveness of exploratory query planning. We discuss each of these below.

6.1 Model Evaluation

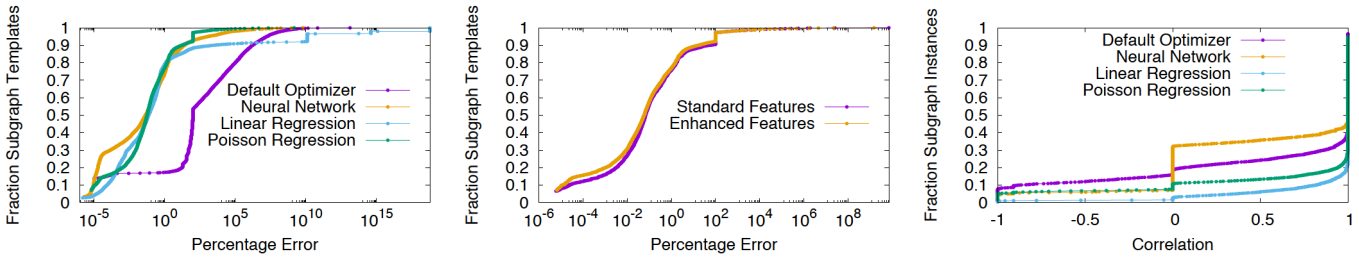
6.1.1 Training

Let us look at the training error of different learning models. Figure 6(a) shows the results over tens of thousands of subgraph templates. We also included the prediction error from the optimizer's default estimation as a baseline comparison. We notice that for 90% of the subgraph templates, the training error of all three models (neural network, linear regression, and Poisson regression) is less than 10%. For the baseline, however, only 15% of the subgraph templates achieve the same level of performance. Therefore, our learning models significantly outperform the baseline.

Figure 6(b) shows the effect of using the enhanced cardinality features on the prediction accuracy. The enhanced features include the square, square root, and log of input cardinality, as discussed in Section 3.3.1, to account for operators that can lead to a non-linear relationship between the input and output cardinality. We observe that adding these features does lead to an improvement in terms of the training accuracy. As we will see in Section 6.2, the enhanced features lead to more (better) query plan changes.

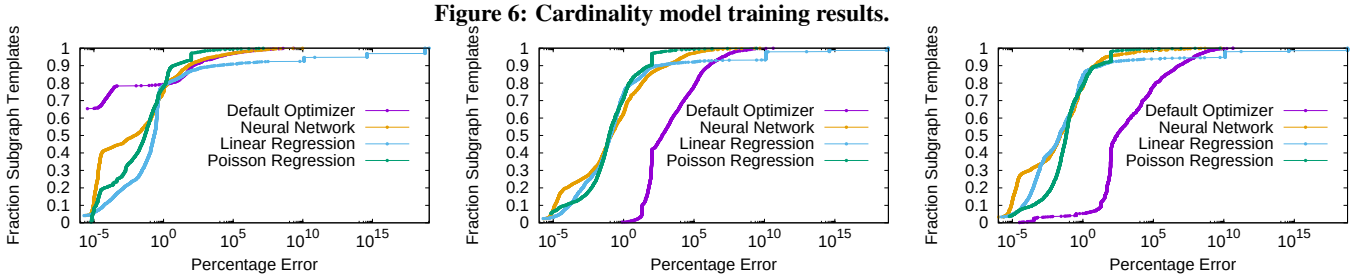
Figure 6(c) shows the Pearson correlation between the predicted cardinality and the actual cardinality for different models. We see that both linear and Poisson regression achieve higher correlation than the baseline. Surprisingly, although neural network attains very low training error, the correlation is lower than the baseline.

In order to study the impact of the root operator on the quality of cardinality estimation, we group the subgraph templates by the type of their root operator. Figure 7(a)– 7(c) shows the training error of our models and the baseline on subgraph templates whose root operators are scan, filter, and hash join. While for the scan operator, the accuracy of the optimizer's estimates is comparable to our models', our models perform significantly better than the baseline for the other two operators. This gives us insight that some models are more important than others, and therefore help us decide which model to cache when we have limited storage budget. For GLM models (Poisson regression), which turn out to be the best, we only need to store the feature name and weight pairs and this results in an average model size of 174 bytes (75^{th} percentile 119 bytes, 90^{th} percentile 418 bytes). We built a total of 34,065 models in this experiment, resulting in a total size of 5.7MB, which is easily manageable. To fit larger number of models within a storage budget, we first filter models to have accuracy improvement (over the optimizer's default estimation) above a minimum threshold and then maximize coverage (number of applicable jobs) while staying



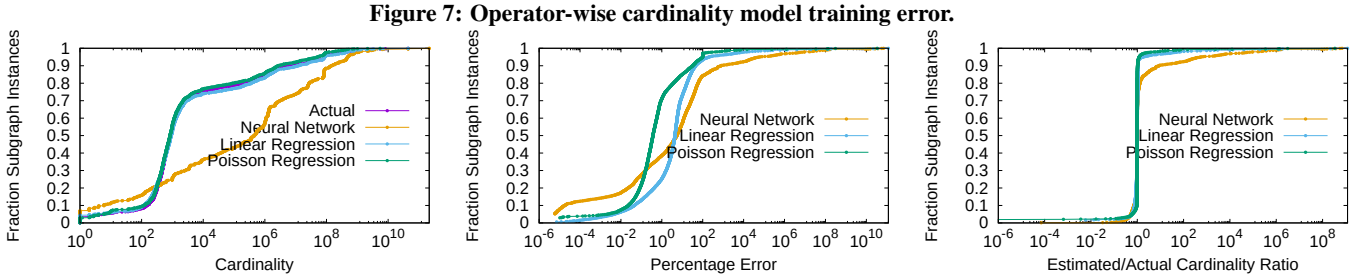
(a) Comparing different models (b) Standard vs Enhanced features (c) Cardinality Correlation

Figure 6: Cardinality model training results.



(a) Scan (b) Filter (c) Hash Join

Figure 7: Operator-wise cardinality model training error.



(a) Predicted vs Actual Cardinality (b) Percentage Error in Cardinality (c) Cardinality Ratio (compare to Figure 1(a))

Figure 8: Cross-validation performance.

within the budget. This maps to the classic knapsack problem [29] that can be solved using dynamic programming.

6.1.2 *Cross-Validation*

We now compare different learning models over ten-fold cross validation. Figure 8(a) shows the cumulative distributions of predicted and actual cardinalities. We can see that both linear and Poisson regression follow the actual cardinality distribution very closely. Figure 8(b) shows the percentage error of different prediction models. Poisson regression has the lowest error, with 75^{th} percentile error of 1.5% and 90^{th} percentile error of 32%. This is a huge performance improvement over the default SCOPE optimizer, which has 75^{th} and 90^{th} percentile errors of 74602% and 5931418% respectively! It is worth noting that while neural network achieves the smallest training error, it exhibits the largest cross-validation error compared to the other two models. This is due to overfitting given the large capacity of neural network and the relatively small observation space and feature space, as also discussed in Section 3.4.

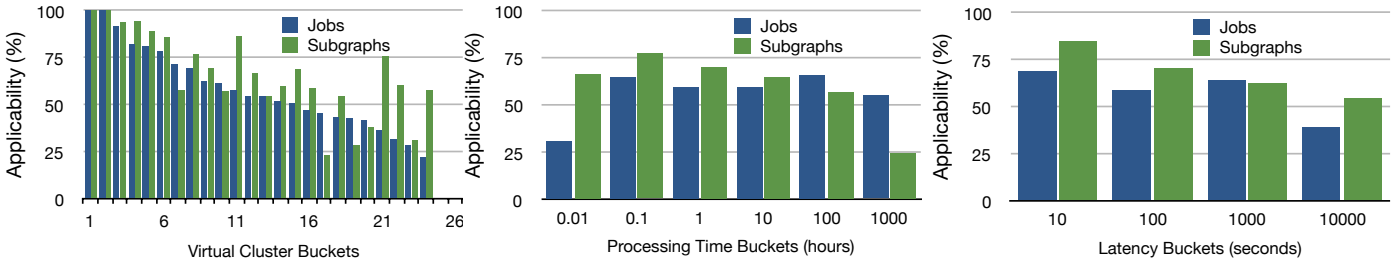
Lastly, Figure 8(c) shows the ratio between our model's predicted cardinality and the actual cardinality for each subgraph. We can see that the ratio is very close to 1 for most of the subgraphs across all three models. Compared to linear regression and Poisson regression, we notice that neural network overestimates 10% of the subgraphs by over 10 times. Again, we suspect that this is due to the aforementioned overfitting. Nevertheless, compared to the same figure (Figure 1(a)) generated using the optimizer's estimation, all of our models achieve significant improvement.

6.1.3 *Applicability*

We now evaluate the applicability of our cardinality models. We define the subgraph applicability as the percentage of subgraphs having a learned model and the job applicability as percentage of jobs having learned cardinality model for at least one of their subgraphs. Figure 9(a) shows the applicability for different virtual clusters (VCs)[3] — 58% (77%) VCs have at least 50% jobs (subgraphs) impacted. We further subdivide the jobs/subgraphs into processing time[4] and latency[5] buckets and evaluate the applicability over different buckets. Figures 9(b) and 9(c) show the result. Interestingly, the fraction of subgraphs impacted decrease both with larger processing time and larger latency buckets. This is because there are fewer number of jobs in these buckets and hence fewer overlapping subgraphs across jobs. Still, more than 50% of the jobs are covered in the largest processing time bucket and almost 40% are covered in the largest latency bucket.

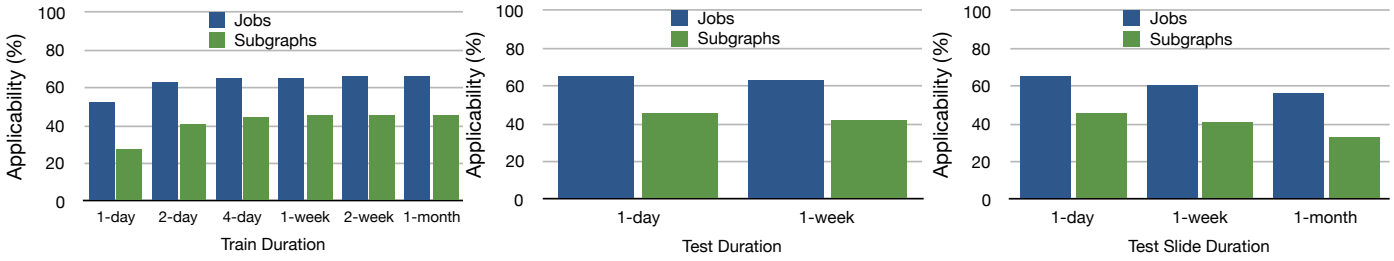
Next, we evaluate how the applicability changes as we vary the training and testing duration. Figure 10(a) shows the applicability over varying training duration, from one day to one month, and testing one day after. We see that two-day training already brings the applicability close to the peak (45% subgraphs and 65% jobs). This is because most of the workload consists of daily jobs and a two-day window captures jobs that were still executing over the day boundary. This is further reflected in Figure 10(b), where the applicability remains unchanged when we vary the test window from a day to a week. Finally, we slide one-day testing window by a week

[3]A virtual cluster is a tenant having an allocated compute capacity and controlling access privileges to its data.
[4]Processing time is the total time of all containers used in a query.
[5]Latency refers to the time it takes for a query to finish executing.



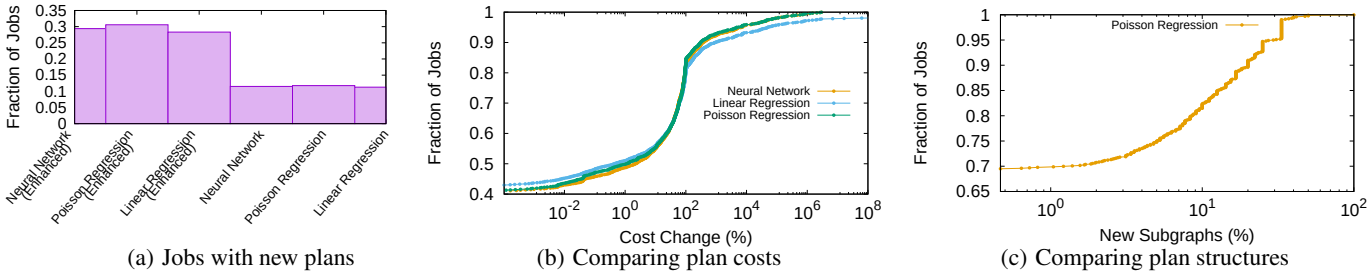
(a) Applicability across virtual clusters (b) Applicability across processing times (c) Applicability across latencies

Figure 9: Job and subgraph applicability of the learned cardinality models over different buckets.



(a) Varying Train Duration (1-day test) (b) Varying Test Duration (1-week train) (c) Sliding Test (1-week train, 1-day test)

Figure 10: Job and subgraph applicability of the learned cardinality models over varying training and testing duration.



(a) Jobs with new plans (b) Comparing plan costs (c) Comparing plan structures

Figure 11: Plan evaluation with the learned cardinality models over production workloads.

and by a month in Figure 10(c). We can see that the applicability drop is noticeable when testing after a month, indicating that this is a good time to retrain our models to adapt changes in the workload.

6.2 Plan Evaluation

In this section, we evaluate how the cardinality models affect the query plans generated by the optimizer. We replay the same workload that was used to train the models in Section 6.1, and predict cardinalities wherever possible using the learned models. The average latency for querying the model server is 14ms, which is insignificant compared to the total compilation latency (on the order of minutes) for SCOPE like batch processing systems. We do not observe overhead on the optimizer for using the learned cardinality model since the features are collected over subexpressions that are already memoized [15] and the cardinality value is computed using a linear combination of the features, bypassing the mathematical calculations for default cardinality estimation and satisfying requirement R3 from Section 2.

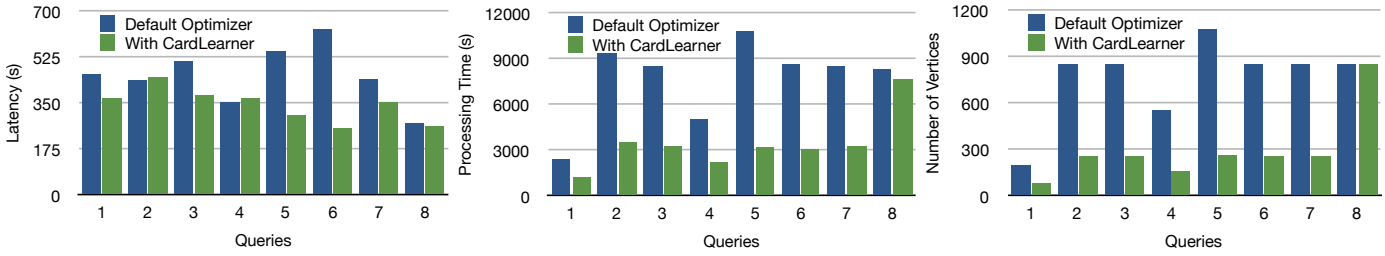
First, we compute the percentage of jobs whose query plan change after applying our models. This is important for two reasons: (i) change in query plan usually implies generation of new subgraph templates, which could be used for further training and improving applicability, and (ii) the new query plan has the potential to significantly improve the performance. Figure 11(a) shows that when training models using standard features (Section 6.1), on average, only 11% of the jobs experience changes in query plans. However, once we incorporate the enhanced features, which capture the non-linear relationships between the input and output cardinality, these percentages go up to 31%. This is significant when considering the hundreds of thousands of jobs per day in SCOPE and the potential savings with better plans. Finally, note that plan changes depend on whether the query costs have changed with better cardinalities or not. In this work, we have kept the cost model intact and focused on improving the cardinalities. Improving cost model accuracy will be a part of future work.

Next, we compute the percentage cost change before and after applying our models to the jobs in the workload. Our results show that 67% of the jobs have cost reduction of more than 10^5, and 30% of the jobs have cost increase of more than 10^3. Figure 11(b) shows the absolute change for all three models. We can see a 75^{th} percentile cost change of 79% and 90^{th} percentile cost change of 305%. Thus, the high accuracy cardinality predictions from our models significantly impact and rectify the estimated costs.

Finally, to understand the changes to the query plans, Figure 11(c) shows the percentage new subgraphs generated in the recompiled jobs due to improved cardinality estimates. We only show for Poisson regression which was seen to be the best earlier. We can see a 75^{th} percentile change of 5% and 90^{th} percentile change of 20%, indicating that the cost changes indeed lead to newer and better query plans for execution.

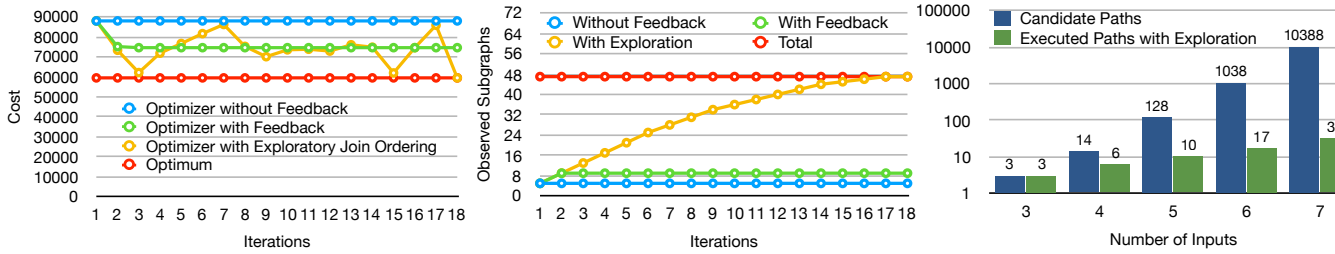
6.3 Performance Evaluation

We now evaluate the performance improvement brought by CARDLEARNER. We consider three metrics for our evaluation: (i) the end-to-end latency which indicates the performance visible to the user, (ii) the processing time which indicates the cost



(a) Query Latency (b) Total Processing Time (c) Total Vertices Launched

Figure 12: Performance evaluation over a subset of production workload, with and without CARDLEARNER.



(a) Plan cost over iterations (b) Subgraph observations over iterations (c) Exploration runs over varying inputs

Figure 13: Evaluating the effectiveness of exploratory join ordering.

of running the queries in a job service [11], and (iii) the number of containers (or vertices) launched which indicates the resource consumption. In this section, we focus on evaluating the learning models discussed in Section 3. We separately present the effectiveness of our exploratory planner in the next section.

Workload. In order to evaluate performance, we need to re-run the production jobs over the same production data. However, production resources are expensive and so it is not possible to re-run all of the tens of thousands of jobs that we analyzed in the previous sections. Therefore, we picked a small subset of the jobs for performance evaluation as follows. We picked a pre-production virtual cluster from all of the virtual clusters that we analyzed, and considered unique hourly jobs having end-to-end latency within 10 minutes (to make sure we do not cause a major noise). Amongst these jobs we picked 8 jobs that process unstructured data, use SPJA operators, and contain a UDO — all of these being typical of production workloads that we see at Microsoft. We executed these jobs with and without CARDLEARNER by directing the output to a dummy location, similar to prior work [2]. We also disabled the opportunistic scheduling [7] to make the measurements from different queries comparable.

Figure 12(a) shows the end-to-end latencies of each of the queries with and without CARDLEARNER. More accurate cardinalities lead all queries, except query 8, to have plan changes. These included changes in physical operator implementations and changes in degree of parallelism. However, the plan changes in queries 2 and 4 are not in the critical path of the job graph, i.e., the changes get masked by slower parallel execution branch, and hence the latency changes are not observable. Other queries such as queries 5 and 6, see a significant performance improvement (around 100%) since their plan changes were in the critical path. Query 5 also has an operator implementation change from sort merge join to hash join, which is preferred for smaller datasets (default optimizer was overestimating heavily). The key reasons why the default SCOPE optimizer suffers from inaccurate cardinality estimation in these 8 queries are: (i) cardinality of unstructured data is estimated using the input size and a constant row length, (ii) the UDOs are treated as black boxes with a constant selectivity, and (iii) in certain cases, even the filter selectivities are approximated using fixed constants since collecting statistics is too expensive at SCOPE-scale. Such magic constants derived from key workloads are typical in commercial database system implementations [35]. Overall, we see an average improvement of 25% in end-to-end latency, which is valuable for reducing costs and improving user experience in SCOPE clusters. Thus, improving cardinality estimates indeed leads to better performance.

Figure 12(b) compared the processing time of queries with and without CARDLEARNER. The improvements are much more significant here because even though the plan change may not be on the critical path of the query, it still translates to lower processing time. Overall, there is a 55% drop in total processing time, which is significant for dollar savings in terms of operational costs.

Finally, Figure 12(c) shows the total number of vertices (containers) launched by each of the queries. Interestingly, the savings are much more significant. This makes sense because we saw in Section 2 how the default optimizer significantly overestimates the cardinalities most of the times, resulting in a large number of partitions to be created and correspondingly large number of containers to be launched, each with a very small amount of data to process. Tighter cardinality estimations, using our models, help avoid the overhead of creating too many partitions and wasting container resources over them. Overall, we launch 60% less vertices using CARDLEARNER. This is highly desirable to (i) improve query performance, (ii) free up spare capacity for other workloads, and (iii) reduce the resource allocation load on the resource manager.

6.4 Exploration Evaluation

We now evaluate the effectiveness of our exploratory query planner. The focus of our evaluation is to show how quickly the algorithm prunes the search space, exploits unobserved subgraph templates, and finds the optimal plan. As discussed in Section 5.6, our current prototype provides the join ordering hints which the users can later enforce in their scripts. Thus, for the purpose of evaluation, we use a synthetic workload along with the unmodified SCOPE cost model, which models the expected latency of queries. Our synthetic workload contains a join query over 6 randomly generated tables with varying cardinalities, where any pair of tables can have a join predicate with probability 0.8, and a random join

selectivity between 0 to 1. The query is replayed multiple times to simulate static workload tuning, as described in Section 4.2.

Figure 13(a) compares the (actual) cost of the plans chosen by the exploratory algorithm against the plans chosen by several alternatives. The default optimizer with bottom-up (Selinger) planner and without statistics feedback picks the same sub-optimal plan every time. When we turn on the feedback, the planner picks a better plan at iteration 2 but does not explore alternate plans in further iterations, and therefore, fails to find the optimal plan. Finally, when we turn on exploration (ranking set to `OPT_OBSERVATIONS`), the planner is able to explore all unobserved subgraph templates in 17 more iterations and find the optimal plan. By design, the exploration might consider more expensive alternatives. However, we feed back the cost of each subplan into the *RuntimeCosts* cache and build models over newly observed subgraphs after each iteration. This helps efficiently prune the suboptimal plans early (Algorithm 1). Thus, in our experiments, exploration never considers plans that are more expensive than those produced by the baseline Selinger planner.

Figure 13(b) shows that while the baseline planner and the planner with feedback can only explore a small subset of the subgraph templates and build cardinality models over them, the exploratory planner is able to quickly cover all subgraph templates in 17 more iterations. The cardinality models built over those newly observed subgraph templates could be useful across multiple other queries, in addition to helping find the optimal plan for the current query.

Figure 13(c) shows the number of executions needed to complete exploration when varying the number of inputs. We see that while the candidate paths grow exponentially, the number of executions required for the exploratory planner to find the optimal plan only grows in polynomial fashion, e.g., just 31 runs for 7 join inputs. The reason is that our exploratory planner picks the plan that maximizes the number of newly observed subgraph templates, which grows polynomially with the number of inputs. Once the planner has built cardinality models for most subgraph templates, it can produce the optimal plan.

During our experiments, we observe negligible difference in the runtime between the baseline planner and the exploratory planner, which is consistent with our analysis in Section 4.1; the exploratory planner only adds a small constant time overhead, still satisfying requirement R3 from Section 2.

We repeated the above evaluation on TPC-H [43] dataset using a query that denormalizes all tables [41, 37]. the exploration planner was able to explore and build all relevant cardinality models using just 36 out of 759 possible runs. In summary, the exploratory planner is able to efficiently prune the search space and explore alternate subgraph templates, thereby avoiding the bias in learning cardinalities and eventually producing the optimal plans.

7. RELATED WORK

Dynamic Query Re-optimization. Dynamic query re-optimization has been studied extensively in the past [19, 28, 8, 46]. The core idea is to monitor the cardinality, data distribution statistics, e.g., max, min, distinct values, and CPU performance counters during the query execution and compare it to the optimizer's estimates. If the estimates are found to be inaccurate, the optimizer uses the observed statistics and the intermediate results to re-optimize the query. Thus, the optimization phase and the execution phase can interleave multiple times during query processing, also referred to as adaptive query processing in commercial databases [38, 12, 32].

Dynamic query re-optimization has three major drawbacks: (i) cardinality estimation errors are known to propagate exponentially and a large portion of the query may have already been executed (using a suboptimal plan) before the system performs re-optimization, (ii) adjusting the query plan, in case of re-optimization, is tedious in a distributed system, since intermediate data needs to be materialized (blocking any pipelined execution), and stitched to the new plan, and (iii) the plan adjustments overheads are incurred for every query, even when portions of query plans overlap across multiple queries, as often observed in production workloads [22, 21, 20].

Therefore, instead of re-optimizing a single query on-the-fly, our approach is based on leveraging feedback, i.e., optimize the current query using the statistics collected from past query executions.

Feedback-based Query Optimization. One of the early works on feedback-based query optimization, LEO [39], considers adjusting the cardinalities for different operators. As discussed in Section 3.1, this approach suffers from the linearity assumption and the lack of a rich feature set to capture the complex cardinality functions in production workloads. More recently, ROPE [2] considered using feedback from both data and code properties to re-optimize queries. The idea in ROPE is to observe and feedback subgraph matches. As a result, ROPE improves over LEO in terms of the accuracy of the feedback, since the exact same subgraphs (and their statistics) are fed back. However, it has much less coverage (applicability) due to strict exact subgraph match requirement. ROPE also requires an online feedback loop, where statistics are collected and fed back as soon as the query finishes. This is tedious in production environments due to extra coordination between the runtime and the optimizer to collect and apply the feedback.

Finally, both LEO and ROPE suffer from learning bias, i.e., they never try alternate plans which have higher estimated cardinalities/-costs but lower actual cardinalities/costs. As a result, they may be trapped in local optima. Proactive monitoring [10] explores alternate execution plans by modifying the optimizer to collect additional statistics when executing the original plan. However, the proposed techniques are only applicable to single-table expressions with filter predicates and join expressions whose join predicates are restricted to primary-key/foreign-key. Hence, a more general approach is required to explore and discover alternate query plans. Moreover, proactive monitoring only benefits the *same* queries that have been executed in the past, again leading to low applicability.

Learned Optimizations. There is a recent trend of applying machine learning techniques to improve different components of a data system. The most prominent being learned indexes [25], an approach to overfit a given stored data and create an index structure that provides faster lookup as well as smaller storage footprint. Other examples include physical design and database administration using learning techniques [45, 1]. In this paper, we leverage machine learning techniques to replace cardinality estimation, one of the core components of a query optimizer.

8. CONCLUSION

Cardinality estimation is a well-known problem in traditional databases. It becomes further challenging in big data systems due to massive data volumes and the presence of unstructured data and user code. In this paper, we presented CARDLEARNER, a radically new approach to learn cardinality models from previous job executions, and use them to predict the cardinalities in future jobs. Our motivation comes from the overlapping and recurring nature of production workloads in shared cloud infrastructures at Microsoft, and the idea is to learn a large number of small models for subgraph templates that overlap across multiple jobs. We introduced the learned cardinality models, presented an exploration technique to avoid learning bias, described the end-to-end feedback loop, and presented a detailed experimental evaluation that demonstrates five orders of magnitude lower error with learned cardinality models.

9. REFERENCES

[1] Predictive Indexing with Reinforcement Learning. In *SIGMOD (to appear)*, 2018.

[2] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Re-optimizing Data-parallel Computing. In *NSDI*, pages 21–21, 2012.

[3] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *EuroSys*, pages 29–42, 2013.

[4] Asimov System. https://mywindowshub.com/microsoft-uses-real-time-telemetry-asimov-build-test-update-windows-9/.

[5] Amazon Athena. https://aws.amazon.com/athena/.

[6] Google BigQuery. https://cloud.google.com/bigquery.

[7] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *OSDI*, 2014.

[8] N. Bruno, S. Jain, and J. Zhou. Continuous cloud-scale query optimization and processing. *Proc. VLDB Endow.*, 6(11):961–972, Aug. 2013.

[9] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2):1265–1276, 2008.

[10] S. Chaudhuri, V. Narasayya, and R. Ramamurthy. A pay-as-you-go framework for query execution feedback. *Proc. VLDB Endow.*, 1(1):1141–1152, Aug. 2008.

[11] Azure Data Lake. https://azure.microsoft.com/en-us/solutions/data-lake/.

[12] Adaptive Query Processing in DB2. https://www.ibm.com/support/knowledgecenter/en/ssw_ibm_i_73/rzajq/rzajqAQP.htm.

[13] W. Gardner, E. P. Mulvey, and E. C. Shaw. Regression analyses of counts and rates: Poisson, overdispersed poisson, and negative binomial models. *Psychological bulletin*, 118(3):392, 1995.

[14] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[15] G. Graefe. The Cascades Framework for Query Optimization. In *IEEE Data Engineering Bulletin*, volume 18, pages 19–29, 1995.

[16] G. Graefe and W. J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *ICDE*, pages 209–218, 1993.

[17] Z. Gu, M. A. Soliman, and F. M. Waas. Testing the Accuracy of Query Optimizers. In *DBTest*, pages 11:1–11:6, 2012.

[18] Y. E. Ioannidis and S. Christodoulakis. On the Propagation of Errors in the Size of Join Results. In *SIGMOD*, pages 268–277, 1991.

[19] Z. G. Ives, D. Florescu, M. Friedman, A. Levy, and D. S. Weld. An adaptive query execution system for data integration. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 299–310, New York, NY, USA, 1999. ACM.

[20] A. Jindal, K. Karanasos, S. Rao, and H. Patel. Thou Shall Not Recompute: Selecting Subexpressions to Materialize at Datacenter Scale. In *VLDB*, 2018.

[21] A. Jindal, S. Qiao, H. Patel, Z. Yin, J. Di, M. Bag, M. Friedman, Y. Lin, K. Karanasos, and S. Rao. Computation Reuse in Analytics Job Service at Microsoft. In *SIGMOD*, 2018.

[22] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. n. Goiri, S. Krishnan, J. Kulkarni, and S. Rao. Morpheus: Towards automated slos for enterprise clusters. In *OSDI*, pages 117–134, 2016.

[23] S. Kandula, A. Shanbhag, A. Vitorovic, M. Olma, R. Grandl, S. Chaudhuri, and B. Ding. Quickr: Lazily Approximating Complex AdHoc Queries in BigData Clusters. In *SIGMOD*, pages 631–646, 2016.

[24] K. Karanasos, A. Balmin, M. Kutsch, F. Ozcan, V. Ercegovac, C. Xia, and J. Jackson. Dynamically Optimizing Queries over Large Scale Data Platforms. In *SIGMOD*, pages 943–954, 2014.

[25] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. *arXiv preprint arXiv:1712.01208v2*, 2017.

[26] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How Good Are Query Optimizers, Really? *PVLDB*, 9(3):204–215, 2015.

[27] SIGMOD Blog. http://wp.sigmod.org/?p=1075.

[28] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic. Robust query processing through progressive optimization. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 659–670, New York, NY, USA, 2004. ACM.

[29] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[30] P. McCullagh. Generalized linear models. *European Journal of Operational Research*, 16(3):285–292, 1984.

[31] J. Neter, M. H. Kutner, C. J. Nachtsheim, and W. Wasserman. *Applied linear statistical models*, volume 4. Irwin Chicago, 1996.

[32] Optimizer Adaptive Features in Oracle. https://blogs.oracle.com/optimizer/optimizer-adaptive-features-in-oracle-database-12c-release-2.

[33] K. Rajan, D. Kakadia, C. Curino, and S. Krishnan. PerfOrator: eloquent performance models for resource optimization. In *SoCC*, 2016.

[34] R. Ramakrishnan, B. Sridharan, J. R. Douceur, P. Kasturi, B. Krishnamachari-Sampath, K. Krishnamoorthy, P. Li, M. Manu, S. Michaylov, R. Ramos, N. Sharman, Z. Xu, Y. Barakat, C. Douglas, R. Draves, S. S. Naidu, S. Shastry, A. Sikaria, S. Sun, and R. Venkatesan. Azure Data Lake Store: A Hyperscale Distributed File Service for Big Data Analytics. In *SIGMOD*, pages 51–63, 2017.

[35] Selectivity Guesses. https://www.sqlskills.com/blogs/joe/selectivity-guesses-in-absence-of-statistics/.

[36] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, 1979.

[37] A. Shanbhag, A. Jindal, S. Madden, J. Quiane, and A. J. Elmore. A Robust Partitioning Scheme for Ad-hoc Query Workloads. In *SoCC*, pages 229–241, 2017.

[38] Adaptive Query Processing in SQL Server. https://docs.microsoft.com/en-us/sql/relational-databases/performance/adaptive-query-processing.

[39] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *PVLDB*, pages 19–28, 2001.

[40] M. Stone. Cross-validatory choice and assessment of statistical predictions. *Roy. Stat. Soc.*, 36:111–147, 1974.

[41] L. Sun, M. J. Franklin, S. Krishnan, and R. S. Xin. Fine-grained Partitioning for Aggressive Data Skipping. In *SIGMOD*, pages 1115–1126, 2014.

[42] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *PVLDB*, 2(2), 2009.

[43] TPC-H Benchmark. http://www.tpc.org/tpch.

[44] I. Trummer and C. Koch. A Fast Randomized Algorithm for Multi-Objective Query Optimization. In *SIGMOD*, pages 1737–1752, 2016.

[45] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *SIGMOD*, pages 1009–1024, 2017.

[46] S. Zeuch, H. Pirk, and J.-C. Freytag. Non-invasive Progressive Optimization for In-memory Databases. *PVLDB*, 9(14):1659–1670, 2016.

[47] J. Zhou, N. Bruno, M.-C. Wu, P.-Å. Larson, R. Chaiken, and D. Shakib. SCOPE: parallel databases meet MapReduce. *VLDB J.*, 21(5):611–636, 2012.