DISTILL: Low-Overhead Data-Driven Techniques for Filtering and Costing Indexes for Scalable Index Tuning

Tarique Siddiqui
Microsoft Research
Redmond, Washington, USA
tasidd@microsoft.com

Wentao Wu
Microsoft Research
Redmond, Washington, USA
wentwu@microsoft.com

Vivek Narasayya
Microsoft Research
Redmond, Washington, USA
viveknar@microsoft.com

Surajit Chaudhuri
Microsoft Research
Redmond, Washington, USA
surajitc@microsoft.com

ABSTRACT

Many database systems offer index tuning tools that help automatically select appropriate indexes for improving the performance of an input workload. Index tuning is a resource-intensive and time-consuming task requiring expensive optimizer calls for estimating the cost of queries over potential index configurations. In this work, we develop low-overhead techniques that can be leveraged by index tuning tools for reducing a large number of optimizer calls without making changes to the tuning algorithm or to the query optimizer. First, index tuning tools use rule-based techniques to generate a large number of syntactically-relevant indexes; however, a large proportion of such indexes are spurious and do not lead to a significant improvement in the performance of queries. We eliminate such indexes much earlier in the search by leveraging patterns in the workload, without making optimizer calls. Second, we learn cost models that exploit the similarity between query and index configuration pairs in the workload to efficiently estimate the cost of queries over a large number of index configurations using fewer optimizer calls. We perform an extensive evaluation over both real-world and synthetic benchmarks, and show that given the same set of input queries, indexes, and the search algorithm for exploration, our proposed techniques can lead to a median reduction in tuning time of 3× and a maximum of 12× compared to state-of-the-art tuning tools with similar quality of recommended indexes.

PVLDB Reference Format:
Tarique Siddiqui, Wentao Wu, Vivek Narasayya, and Surajit Chaudhuri. DISTILL: Low-Overhead Data-Driven Techniques for Filtering and Costing Indexes for Scalable Index Tuning. PVLDB, 14(10): 2019 - 2031, 2022.
doi:10.14778/3547305.3547309

1 INTRODUCTION

Given a workload and a set of constraints (e.g., a storage budget), index tuning tools [13, 21, 35] recommend a set of appropriate indexes for improving the performance of the workload. However, *scaling* these tools to a large workload remains a challenging task. While this is already a problem for on-premises databases, the scalability challenge is further amplified today in the cloud environment where the cloud vendors need to tune a large number of databases (typically on production servers), thereby adding to the resource and cost overheads [16].

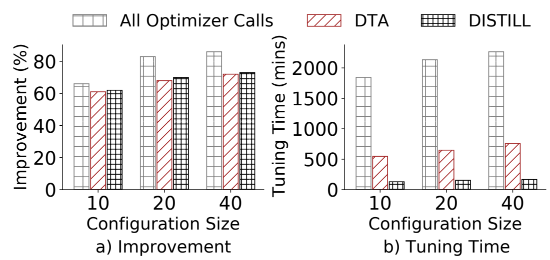


Figure 1: Impact of DISTILL on performance improvement and tuning time on TPC-DS workload consisting of 910 queries

Given an input workload, index tuning tools first generate syntactically-relevant indexes for each query and then search for the best configuration among the generated indexes via configuration enumeration. During enumeration, what-if optimizer calls [14] are used which help estimate the cost of a given (query, index configuration) pair without building the indexes. Each call is at least as expensive as a regular optimizer call, consuming a significant (over 70%) fraction of the tuning time. For large workloads, these calls significantly increase the tuning time as well as the CPU and memory burden on the DBMS. For scalability, it is therefore crucial to employ techniques for identifying when what-if calls can be avoided without affecting the quality of recommended indexes.

While there exist techniques for *reducing* the search space [6, 12, 13, 33, 37], making what-if calls for every (query, configuration) pair remains challenging even for such reduced search spaces. For example, the *greedy* algorithm [13] used by state-of-the-art tools [6] requires $O(mnk)$ what-if calls, where m is the number of queries in the input workload, n is the number of candidate indexes, and k is the maximum configuration size. In this work, we develop complementary techniques that further reduce optimizer calls and can be incorporated into index tuning tools *without* making changes to the search algorithm or to the query optimizer.

First, we observe that index tuning tools generate indexes based on syntactic relevance by applying rules on indexable columns (e.g., filter, join, group-by, and order-by) [13, 21]. For each syntactically-relevant index, a what-if call is made to identify whether it can improve the performance of the query. Unfortunately, a large percentage of such indexes (e.g, about 50-70% on our evaluated workloads) are *spurious*, i.e., given a query and an index, the optimizer either does not use the index or uses it with minimal gain (e.g., < 5%

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 14, No. 10 ISSN 2150-8097.
doi:10.14778/3547305.3547309

improvement in cost). To address this, we capture signals for such spurious indexes using information available in the physical plans of queries on existing physical design, index structures, and tables. We pre-train a *workload-agnostic* filtering model, called Index Filter, that automatically learns patterns over these signals and can be used during index tuning for filtering spurious indexes over any databases. We show that Index Filter can be accurately trained over a small number of databases (e.g., 3 to 4), and is extremely efficient to use with multiple orders of magnitude lower inference time compared to what-if calls.

Despite pruning, the number of optimizer calls during enumeration can still be considerable. We therefore develop techniques that replace a significant number of optimizer calls (which can be expensive) for costing (query, configuration) pairs with cheaper cost models trained for each workload. It is challenging to develop a general technique for all databases and queries due to large varieties in schema, query structures, and data distributions. However, many queries in large workloads are typically similar [25, 32], e.g., multiple instances of the same query template or stored procedure parameterized differently. Furthermore, many indexes explored during tuning are also similar, e.g., having the same prefix of key columns, or influencing the same set of operators in the plan, resulting in similar cost reductions. As a result, we see that the number of unique costs is often much smaller than the number of index configurations explored during tuning (e.g., on average only 6 unique costs over 81 configurations explored per query for the TPC-H workload). To leverage these characteristics, we group similar queries and learn a query template- and index-specific cost model (in short Index Cost Model) for each group separately. For efficient in-situ training during the tuning, we develop an iterative training procedure (with optimality guarantees) and select diverse training instances (e.g., queries with different selectivities, indexes affecting different operators in the query) that minimize the number of optimizer calls for training each model (e.g., < 50 optimizer calls per model on average across workloads). Like Index Filter, Index Cost Models are also significantly more efficient than the what-if calls.

There has been related work [8, 9] that *instrument* the optimizer for encoding a compact representation of the optimization search space for a query, which is then used to estimate the cost of multiple indexes. However, these techniques require invasive changes to the optimizer. There have also been cost-derivation techniques [13, 22, 28] that make optimizer calls to obtain costs for a few (query, configuration) pairs, and apply rules on obtained costs to derive costs of other configurations. For instance, Database Tuning Advisor (DTA) [6] makes optimizer calls for atomic configurations [12], and derives the costs of larger configurations by taking the minimum cost across subsets of atomic configurations. We observe that these techniques may (a) still need a substantial number of optimizer calls (2× to 5× more than Index Cost Models), (b) require changes to the search enumeration algorithm, e.g., [12] only works with the bottom-up greedy algorithm, and c) are less effective in capturing complex index interactions (i.e., multiple indexes simultaneously improving the performance), thereby resulting in lower quality recommendations than Index Cost Models.

We have implemented the Index Filter and the Index Cost Models in a prototype, called `DISTILL`, that reuses the candidate index generation and the greedy search enumeration steps proposed in [13]. Our evaluation shows that DISTILL helps scale the tuning process to large workloads without sacrificing the quality of recommended indexes. For instance, as depicted in Figure 1a, for a TPC-DS workload of over 900 queries, DISTILL gives similar quality index recommendations as DTA [6] (with unbounded tuning time budget) and a variant of [13] that makes optimizer call for every unique (query, configuration) pair explored during enumeration in 4× and 15× less time respectively. We give an unbounded time-budget to DTA to ensure that the evaluated space of (query, configuration) pairs is similar for all techniques as well as to avoid the influence of other optimizations such as workload compression which are complementary to the techniques discussed in this work.

Contributions. Our contributions can be summarized as follows:

- We discuss the scalability challenges in index tuning and outline a number of opportunities for improvement.
- We develop a workload-agnostic index filtering technique that captures patterns over query structure, statistics, and index structure to identify indexes that do not lead to a significant improvement in costs. We use this technique to remove a large number of syntactically-relevant indexes without affecting the quality of index recommendation significantly.
- We learn index-specific cost models (one for each group of similar queries) on-the-fly during the tuning process and use them to *predict* the costs of other similar (query, configuration) pairs, thereby avoiding many optimizer calls. We develop an iterative training procedure for efficiently training of such models using a small number of optimizer calls.
- We perform a thorough evaluation of DISTILL over multiple synthetic benchmarks and real workloads. Our results show that DISTILL results in similar improvement in performance as making all optimizer calls and unbounded DTA, but with a median and maximum reduction in tuning time of 6× and 20× respectively when compared to making all optimizer calls, and 3× and 12× when compared to DTA.

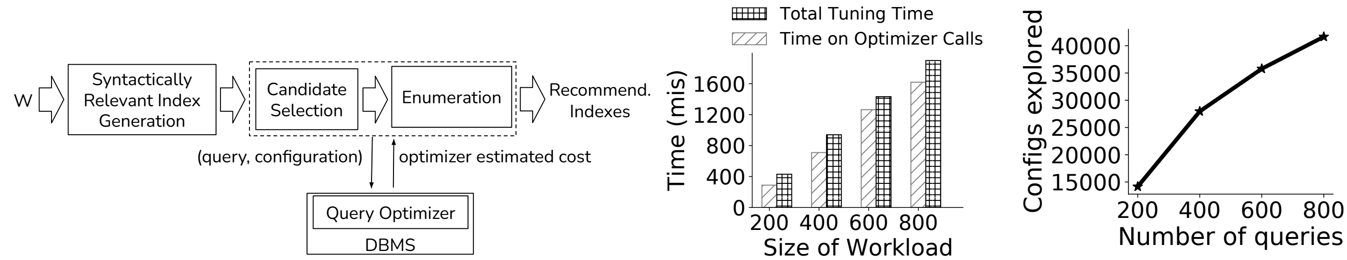
2 BACKGROUND

2.1 Overview of Index Tuning

Figure 2a depicts the typical architecture of an index advisor as described in [13]. Index advisors take as input the workload on a specified database, with certain constraints such as the maximum number of indexes allowed and storage budget. The workload is parsed to extract relevant columns (called indexable columns), i.e., columns that are part of the filter and join predicates, or group-by and order-by clauses. Index advisors then generate syntactically-relevant indexes using indexable columns by applying different rules. Next, for each query, index advisors perform *candidate selection* to identify a subset of useful indexes that lead to a significant improvement in performance of the query given the constraints. A *configuration enumeration* module then searches the space of subsets of candidate indexes (from all queries) and picks a configuration (i.e., subset) that results in the maximum improvement on the workload. In order to estimate the improvement for a given (query, configuration) pair during candidate selection and enumeration, index advisors rely on what-if calls [14], which is an extended functionality of the query optimizer that can estimate the cost of a query given a configuration *without* building the indexes.

2.2 Scalability Challenges

The scalability of an index advisor depends on the number and complexity of queries in the workload, which in turn determine the



(a) A typical architecture of index advisors [13]

(b) Tuning time on varying workload size (TPC-DS)

(c) Configurations explored on varying workload size (TPC-DS)

Figure 2: Architecture of index advisors and scalability challenges

number of configurations enumerated, and the number of what-if optimizer calls made during tuning.

Figure 2b depicts the increase in tuning time for a state-of-the-art index advisor [2] as we increase the number of queries in the TPC-DS workload. As we can see, the tuning time grows significantly as we increase the size of the workload. This is primarily because the space of configurations to explore increases (Figure 2c), with each configuration requiring an expensive optimizer call (consuming 70% to 80% of the overall tuning time). Workloads generated by cloud applications can be even larger and consist of more complex queries (e.g., in the order of hundreds of thousands of statements) [16]; and tuning of such workloads within reasonable amount of time is challenging. Furthermore, the process of tuning a large workload can impose significant overhead on the server being tuned since the physical design tool needs to potentially make many what-if calls to the query optimizer component. It may also affect the performance of workload being concurrently served. While one can use a "B-instance" for index tuning, it comes with a high operational cost and is not practical at scale for service providers that host millions of databases [16]. As a result, in practice index tuning is primarily performed on the production server, imposing a significant overhead in terms of time and resources reserved for production queries.

2.3 Problem Formulation

We consider an input workload $W = \{q_1, q_2, ..., q_n\}$ consisting of n queries that needs to be tuned to select k indexes. We assume that the physical plan for each query that is generated by the optimizer with the existing physical design (i.e., without adding or removing indexes) is provided as part of the workload. We observe that most DBMSs expose functionality to collect historical workload information including the physical plan for a query, e.g., Query Store [4] in Microsoft SQL Server. Such information can be leveraged by DISTILL for analyzing queries without making optimizer calls. Let $C(q_i)$ be the optimizer-estimated cost of q_i with existing physical design, and $C(W)$ be the optimizer's estimated cost for the entire workload W with $C(W) = \sum_{i=1}^{n} C(q_i)$.

Let I be a set of k indexes selected by an index tuner on tuning W, and $C_I(q_i)$ be the optimizer-estimated cost of q_i and correspondingly $C_I(W)$ for the workload W, when using the (hypothetical) indexes in I. The expected performance improvement of W due to I is captured using the notion of "improvement," as defined below:

Definition 1 (Improvement)**.** *Improvement, Δ, is defined as the* decrease *in the cost of the workload W when using the indexes in I, i.e., $\Delta = C(W) - C_I(W)$.*

In this work, we focus on reducing the tuning time with minimal degradation in the improvement of the workload. As discussed earlier, optimizer calls consume most of the index tuning time, thereby posing a scalability challenge. Thus, to reduce tuning time, we aim to replace a large proportion of optimizer calls with significantly more efficient techniques for estimating (query, configuration) costs. We do not make changes to the *configuration enumeration* (or search) component of the index tuning process outlined in Figure 2a. While various implementations have been proposed (e.g., [8, 10, 13]), we chose the classic *greedy* search algorithm, which is efficient and has been used by both *AutoAdmin* [13] and DTA [6]. Recent work has also shown that this greedy algorithm yields state-of-the-art performance [26]. That said, our developed techniques will not leverage any property of the greedy algorithm and hence can be used with any other search algorithm. Formally,

Given an input workload W consisting of the physical plans of the queries generated by the optimizer with the existing physical design on the target database, the number of indexes k to select, and a search algorithm for enumerating index configurations, our goal is to reduce the number of optimizer calls such that we maximize the improvement and minimize the tuning time.

There are a number of possible solutions. First, as an ideal solution for maximizing improvement, we can use the greedy algorithm that makes optimizer calls for *every* (query, configuration) pair enumerated during tuning. However, as noted earlier, this is costly, resulting in high tuning time. Another approach is to use cost-derivation techniques [13] to reduce optimizer calls for a subset of (query, configuration) pairs used by state-of-the-art tools such as DTA [6]. Although such techniques help reduce the tuning time, the number of optimizer calls may still be large. Furthermore, rules applied for cost derivation do not effectively capture the interactions between multiple indexes simultaneously improving the performance, thereby degrading the quality of recommended indexes especially over complex workloads. In this work, our goal is to explore techniques that reduce optimizer calls such that we achieve both high improvement and low tuning time simultaneously. Moreover, for simplicity, we use the size of the final index configuration (k) to be returned as a tuning constraint. However, our techniques do not depend on tuning constraints and can be used with other constraints such as storage budget.

3 OVERVIEW OF OUR SOLUTION

To reduce the number of optimizer calls, we develop a two-step solution depicted in Figure 4. In the first step, we prune out many (query, index) pairs that have no or small (e.g., < 5%) reduction in cost using an workload-agnostic model that is trained offline.

Next, we train index-specific cost models for each group of similar queries in the workload in-situ during the tuning process (using a small number of optimizer calls), and use them to replace a large number of optimizer calls for other similar (query, configuration) pairs. We describe the motivation and details of the steps below.

As discussed earlier, index advisors generate syntactically-relevant indexes by applying rules on indexable columns. Figure 3 depicts the improvement in cost for different fractions of (query, syntactically relevant index) pairs for four synthetic and real workloads (we provide more details on workloads in §6). As we can see, between 60% to 70% of syntactically-relevant indexes do not result in significant changes in costs of queries. Thus, the optimizer calls made on such indexes are unnecessary. To address, we learn offline a workload-agnostic model that uses structure and statistics in the input (query, index) to identify when the index may not lead to a significant improvement in cost. We use this model to remove a large number of spurious candidates (step 4) online during the tuning. Our key insight is that we can probe the original physical plan of query (i.e., the plan generated with existing physical design structures and without adding or removing indexes) to estimate the potential for improvement in the cost of the query due to a given index. For instance, if the join or sort operation is already efficient due to an extensive filtering from earlier operators, adding an index that optimizes subsequent operations is less beneficial. Similarly, if a filter column is not selective, we can easily prune an index with such column as the leading key column. Furthermore, in many cases, we can look at the shape of the query plan (e.g., ordering of physical operators within the original plan) as well as the size of table and types of existing indexes on the table to prune indexes which may not further improve the performance. We capture such signals and automatically learn rules on the signals using a workload-agnostic regression model, called Index Filter (step 1 and 2 in Figure 4). We show that such a model can be accurately learnt using (query, configuration) pairs generated from 3 to 4 databases and workloads. We find that Index Filter can remove over 70% of the spurious indexes with low (typically $< 10\%$) false negatives.

Despite pruning, the enumerated number of configurations resulting from the unpruned indexes may still be large. For instance, we observe that Index Filter reduces the number of syntactic indexes from about 11k to 7k for a TPC-DS workload consisting of about 900 queries. However, the number of resulting configurations explored during the search is over 33k. Making an optimizer call for each one of these configurations is expensive. Furthermore, we find that it is extremely hard to learn an offline model that can accurately predict the cost of an arbitrary (query, configuration) pair for a given database, let alone an unseen database. However, queries in large workloads typically have high degree of similarity in terms of structure [25], e.g., templatized queries, such as stored procedures, that differ only in parameter bindings are common in real-world applications. Similarly, indexes share similarity in terms of key columns or how they influence plans. These properties allow us to train an index-specific cost model model, called Index Cost Model, for each group of similar queries and configurations in-situ (or online) during the tuning (step 5).

A key challenge in learning Index Cost Models is minimizing the number of training instances (i.e., optimizer calls) required as training time adds to the tuning time. To minimize under- or over training, we develop an iterative training procedure with optimality guarantees that starts with a small and *diverse* set of training

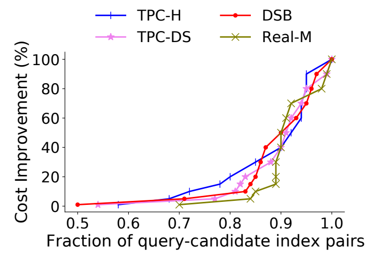


Figure 3: Cost improvement for different fraction of query and syntactically relevant index pairs

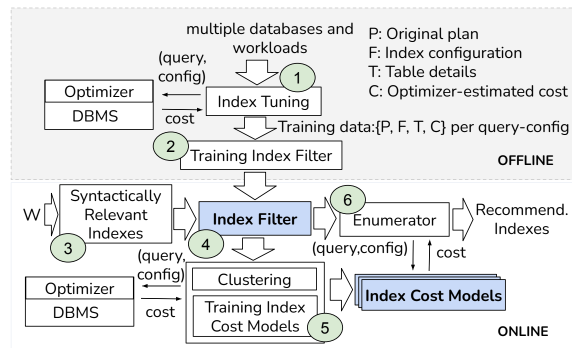


Figure 4: Overview of our solution

instances (sampled from (query, index) pairs selected by Index Filter), and incrementally increases the training size until the model error reduces within a small error threshold. For groups, where the estimation errors of models do not reduce quickly (i.e., within certain fraction of training instances), we fall back to the optimizer calls for estimating costs. On average across workloads, we find that we can train an Index Cost Model using only 30 to 50 (query, configuration) pairs. Once trained, Index Cost Models are cached and used for reducing optimizer calls for several hundreds of (query, configuration) pairs per group during enumeration (step 6).

A key design requirement for DISTILL is that pruning spurious indexes and estimating costs must be much faster than what-if calls, otherwise the purpose of reducing optimizer calls is negated. Furthermore, while the Index Filter is trained offline, the training of Index Cost Model is done during tuning and needs to be efficient as discussed above. To ensure these requirements, the employed machine learning techniques must have low-overhead. We find that tree-based ensemble models with a few tens of trees can reasonably meet these requirements. Furthermore, we apply domain understanding on how indexes improve the performance of queries to reduce the number of features. Overall, the models give high accuracy with extremely fast inference time, i.e., over two orders of magnitude lower inference time compared to what-if calls.

4 FILTERING SPURIOUS INDEXES

Index tuning tools use syntactic analysis, i.e., rules that combine indexable columns from operators such as filter, join, group-by and order-by, to identify a set of indexes for tuning. As depicted in Figure 3, a large number of syntactically relevant indexes are spurious, i.e., they do not result in significant improvement in performance of the queries. In this section, we discuss low-overhead techniques to prune such spurious indexes without degrading the quality of indexes. In particular, we capture four types of signals that are indicative of spurious indexes. However, it is challenging to construct rules that capture interactions between these signals for identifying spurious indexes. To address, we train a workload-agnostic regression model, called Index Filter, that automatically learns rules over the signals using a large corpus of (query, configuration) pairs generated from multiple databases. We first discuss the four signals and then give an overview of how we learn the model.

4.1 Estimating Potential Improvement

While selecting an index for tuning, index tuners ignore the potential for improvement in cost of the plan due to the index. We observe that for many syntactically-relevant indexes, the costs of operators they improve is often significantly smaller compared to the overall cost of the plan. Thus, the improvement in performance due to such indexes is small.

To address, we estimate the potential for improvement for each syntactically-relevant index using the original plan of the query, i.e., the most efficient physical plan obtained by making an optimizer call for the query with the existing physical design of the database. Since our goal is to identify spurious indexes instead of accurately estimating the new cost when using the index, we make simplifying assumptions which help *efficiently* estimate the potential improvement. Specifically, we avoid exploration of alternative join ordering or transformations that cause significant changes in the original plan. An extensive exploration of alternative plans is expensive (equivalent to making optimizer calls) and negates the purpose of index filtering. We therefore make the following assumption.

Assumption 1. *A index has a high potential for improvement if it reduces the cost of operators in the original plan that have high costs relative to the rest of the operators in the plan, and vice-versa.*

An index can reduce the cost of an operator if it satisfies one or both of the following properties:

Property 1 (Filtering). *An index satisfies the filtering property if it helps skip access of one or more tuples during scan, filter, or join operations.*

Property 2 (Interesting Order). *An interesting order is a tuple ordering specified by the columns in a query's join, group-by, or order-by clause. An index covers an interesting order if its key columns are sorted according to that interesting order.*

To estimate the potential improvement, we traverse the original plan in a bottom-up manner, and look for the presence of physical operators whose cost can be reduced if the index was selected. We consider the physical operators (called relevant operators) corresponding to the following logical operators: scan, join, group-by, and sort (or order-by) as depicted in Table 1. The potential for improvement is estimated as the sum of the costs that is reduced for each of the relevant operator in the original plan. We capture this notion via "utility," defined as follows:

Definition 2 (Utility of an Index)**.** *Utility of an index is the sum of the estimated reduction in costs of relevant physical operators in the original plan due to the index, normalized by the total cost of the original plan.*

We normalize to give low importance to indexes that may not lead to substantial improvement in the cost of query, and vice-versa.

Estimating reduction in cost of operator due to an index. We estimate the reduction in the cost of operator using its cost (denoted by C) in the original plan and the average selectivity (denoted by S) of key columns in the index that are used by the operator.

Table 1: Logical and Physical Operators considered for Utility Computation

Logical Operators	Physical Operators
Scan	Table Scan, Index Scan, Index Seek, Clustered Index Scan, Clustered Index Seek
Filter	Filter, Bitmap
Join	Nested-Loop, Hash Match, Merge Join
Group By/Aggregation	Hash Match, Stream-aggregate, Sort
Sort	Sort

Filtering: If the index satisfies the filtering property, the improvement is computed as $(1 - S) \times C$, i.e., the reduction in cost is proportional to the fraction of tuples filtered using the index. In other words, the improvement is high if the cost of the operator is high and the selectivity of key columns is low. The join selectivity can be estimated as follows: Let O be the output cardinality of the join operator in the original plan, and L and R be the input cardinalities to the join. We compute the join selectivity as $J = \frac{O}{L \times R}$. Assuming that each of the left and right input contribute equally to the join output, the selectivity of each of indexable columns is $\sqrt{J}$. We estimate the reduction in cost when adding an index to one of the inputs as $(1 - \sqrt{J}) \times C$. While this estimation approach may be inaccurate when the index causes significant transformations in join ordering or operators, we find it to be effective in identifying cases when the index does not cause any change (typically when J and C are already small).

Ordering: When there is a sort operator in the plan that provides the same ordering of tuples as the index, the improvement is equivalent to the cost of the sort operator. In other words, we assume that the index can help get rid of the sort operator in the plan.

Hash-aggregate to stream-aggregate transformation. We also consider a simple transformation between hash-aggregate and stream-aggregate which happens frequently on adding indexes. If the original plan chooses a hash-aggregate instead of the stream-aggregate, it is likely because the sorting required for the stream-aggregate is more expensive than the hash-aggregate. Thus, if the index already provides the ordering property, the improvement is equal to the cost of hash-aggregate assuming that the cost of computing aggregates in a streaming manner is negligible compared to the cost of sorting.

Examples. Consider a SQL query as depicted in Figure 5a. Figure 5b depicts the selectivity of predicates in the query, and Figure 5c depict the original plan of the query. We first consider a clustered index on column `LCol1` on LineItem table (Figure 5d). We traverse the plan bottom-up, and see that we can replace table scan with clustered index scan; however there is no opportunity for filtering and hence the estimated reduction in cost is 0. Next, we can see that hash-aggregate can be replaced with a stream-aggregate since the index provides the ordering on the group-by column. Since sorting is the most dominant operation while using stream-aggregate, we assume the cost of stream-aggregate is negligible compared to hash-aggregate. Thus, the reduction in cost is 80. Finally, we see that the selectivity of the hash join is 0.50, and the index has the join key as the key column. As discussed above, the reduction in cost of hash join by adding an index on one of the join columns is computed as $(1 - \sqrt{0.50}) \times 160 = 48$. Overall, the total cost reduction is 112 out of 400, with utility as .28. As another example, we consider a clustered index on `OCol2` on the 'Order' table (Figure 5e). In this case, the cost of scan is reduced by a fraction proportional to the selectivity of the predicates, i.e., 54. Besides scan, the index does not reduce the cost of other operators. Thus, the utility of this index is 0.23.

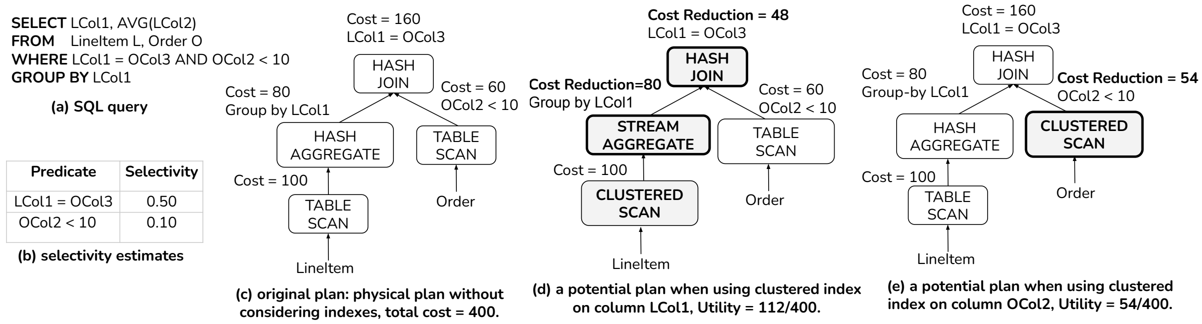


Figure 5: Illustrative examples to show how potential improvement is computed (d and e) over the original plan (a)

While utility is the most influential signals in identifying spurious indexes (§6), we find utility by itself is not sufficient especially when the physical plan generated due to the index is significantly different from the original plan. Thus, we derive three additional signals to identify spurious indexes, as discussed next.

4.2 Shape of Queries and Indexes

In many cases, optimizers apply transformation rules (e.g., aggregate pushdown) that can impact the effectiveness of an index. More generally, query optimizers push more selective operations or ones that reduce intermediate data size down in the plan which help reduce the cost of operators higher in the plan. Hence, building an index that can improve the performance of such lower-level operations is typically more beneficial than ones that affect the higher-level operators. As an example, we observe that a large majority (> 70%) of syntactically-relevant indexes with order-by or group-by columns as leading key columns are spurious. For many such cases, the downstream (i.e., operators below the group-by or order-by) operators such as filter or join operators are highly selective. However, in some cases, when the optimizer pushes an aggregate operation below the join, an index that satisfies the ordering property required for the aggregate may be more beneficial.

To capture such optimizations, we extract the sequence of logical operators (i.e., scan/filter, join, aggregate and sort) for each table in the original plan in a bottom-to-top order. We call each sequence a shape. For instance, the shape of the sub-query on Lineitem table in Figure 5c is scan->aggregate->join where -> depicts the sequence in which tuples are processed. Similarly, the shape of the subquery on the Order table is scan->join. If the same operation occurs twice in a sequence, we only capture the first occurrence to avoid creating a large number of possible shapes.

We also extract the shape of the index using the ordering of key columns in the index. Index tuning tools use rules to combine indexable columns to construct indexes. For instance, Table 2 depicts a set of rules similar to the ones used by DTA [6]. Thus, we capture the shape of an index via the rule that is used for generating it. In general, we find that the more similar the index and subquery shapes, the higher is the impact of an index. Note that an index can have more than one shape if the ordering of key columns satisfies different orderings of operations, e.g., when the same column is used across multiple operations. In the feature vector for learning regression model (described shortly), each possible shape for a subquery or a an index is represented via a feature.

4.3 Physical Operators

We also capture the physical operators listed in Table 1. We set the value to 0 if the index is not relevant to the operator. On the other hand, if relevant, we compute the value using the statistics of the key columns in the index. Specifically, if the physical operator is a filter, scan, or join operator, we assign it a value equal to the average selectivity of the key columns in the index that are used by the operator. If the physical operator is grouping or sort, we assign it a value equal to the average density of the key columns in the index. Density is a function of number of unique values of the column(s), which impacts the cost of grouping and sort operations [5]. We observe that physical operator signals interact with shape-based signals to improve the accuracy of identifying spurious indexes.

Table 2: Different possible rules to combine indexable columns to generate indexes.

S1	selection
S2	join
S3	selection → join
S4	join → selection
S5	order-by → selection → join
S6	group-by → selection → join
S7	order-by → join → selection
S8	group-by → selection → join

4.4 Optimizer Behaviour

We also consider additional properties that help capture the behavior of the query optimizer. For example, we notice that the optimizer may not select more than a certain number of indexes per table, or the optimizer may not build an index on smaller tables (E.g., Nation table in TPC-H) consisting of fewer pages than a specific threshold. Specifically, the properties that we capture include the number of pages, whether there is a clustered indexes already present, the number of non-clustered indexes already present, and whether a scan or filter operation uses a bitmap.

4.5 Learning Rules for Index Filtering

Given an original plan for a query and an index, we learn a regression model that predicts how likely the index will lead to a change in cost of the plan. We find log-transformed labels and using the mean square error as the loss function to be effective. Since query optimizers can often have a small variance in estimated costs across multiple invocations even when the index is not used, we consider a cost change as significant if it is above a threshold α. We use a small threshold of 5% for a low false negative rate, although prior work [18] have used an even higher threshold of 20%. Thus, all indexes with predicted value $\leq$0.05 are pruned.

An alternative formulation is to learn a *classifier* that predicts *whether* or not the plan or cost will change significantly. While we find little difference in the results between the two tasks; for a classifier, a change in the value of α requires retraining, which can be costly. Another issue is less flexibility in controlling the *false* *positive* and *false negative* rates—a high α leads to high false-positive rate and low false-positive rate, and vice-versa.

To generate the features, we probe the original plan of the query using the index to capture the four types of signals discussed in the earlier sections. Observe that the our featurization differs from prior work on plan featurization (e.g., [18]) in that we featurize each plan using the index as context, since we are only interested in parts of the plan that may be influenced by the index. This in turn results in much fewer features, since often only a small set of operators are affected. Given these features, we experimented with multiple machine learning (ML) models including linear models, decision trees, ensembles of trees, and multi-layer perceptron (MLP) neural networks (see §6). Considering both inference time and model accuracy, we find that tree-based ensemble models work the best. In particular, we observe that random forests with 40 trees and a depth of 10 gives a reasonable performance.

Offline training. We train the model offline using multiple databases and workloads, capturing a total of about 75 features. On average, it takes between 80 and 130 hours to generate the labelled training data (by making optimizer calls) using the databases and workloads listed in Table 3, consisting of a maximum of 170k (query, configuration) pairs and their optimizer estimated costs. The training takes less than 5 minutes for the tree-based ensemble models while about 20 minutes for neural network models (only using CPU). The inference time for tree-based ensemble models is about 10 ms (about 1.5 to 2 orders of magnitude faster than optimizer calls) while for MLP-based models, it is in order of 100s of milliseconds.

5 INDEX COST MODELS

Despite pruning, the number of optimizer calls during the tuning may still be large. In particular, if m is the number of queries in the input workload, n is the number of candidate indexes, and k is the maximum number of desired indexes, the number of configurations enumerated during the search is $O(mnk)$. For instance, in our evaluated workloads, we find that greedy algorithm enumerating between 5× to 10× times more configurations than the input set of candidate indexes for $k = 20$.

In order to further reduce the number of optimizer calls, tools such as DTA apply cost derivation techniques [13] that derive the costs of large configurations using the cost of smaller atomic configurations, e.g., by taking the minimum cost among all atomic configurations. While such derivation techniques are partially helpful in reducing the optimizer calls (almost by half over the workloads we evaluated in §6), making optimizer calls for the remaining configurations is still time-consuming and resource-intensive. Further, cost derivation techniques ignore the potential interactions between indexes and hence suffer from low accuracy on complex workloads.

In this section, we explore ML techniques for costing (query, configuration) pairs. While a general model for all possible (query, configuration) pairs is challenging, large workloads typically consist of queries which are similar, e.g., many instances of the same query template [25, 32]. This allows us to train a specialized cost model, called Index Cost Model, for each group of query instances belonging to the same template. We show that we can use a small number of optimizer calls (typically < 50) in-situ during the tuning to accurately train such cost models (thereby avoiding the issue of generalizability across workloads) and use them to estimate the costs of large numbers of (query, configuration) pairs in two orders of magnitude less time than what-if optimizer calls. In the following, we first discuss how we learn Index Cost Models, and then give an overview of an iterative training procedure that we use for efficiently training such models in-situ during the index tuning.

5.1 Learning Index Cost Models

Given a query and an index configuration, we learn an Index Cost Model that returns the estimated cost of the query when using the index configuration. We learn one Index Cost Model for each set of query instances with the same template. Two queries have the same template if they only differ in the parameter bindings. Since all query instances for a given Index Cost Model have the same structure, we use only parameters and configurations for featurization. Furthermore, we construct features in a schema-agnostic manner that helps minimize the number of features and captures similarity across similar indexes. For instance, different indexable columns with similar selectivity and affecting the same set of operators will have a similar set of feature values.

Selectivity of parameters. For each parameter in the query instance, we estimate its selectivity. We also considered using parameter values as features; however, we do not see any significant improvement in accuracy when using parameter values.

Configuration. A configuration is a set of indexes. Each index can have multiple columns. We consider each column, and construct a feature depicting the index type, whether the column is a key column, the type of operator among {scan, join, order-by, group-by}, the highest column position across any index in the configuration if it is a key column (this helps avoid combinatorial explosion in the number of features while still capturing the importance of a column), and the sort order of the columns. Similar to the physical operator signals for Index Filter, we set the feature value to the selectivity if the indexable column influences a filter or join operation, to the density of the column if it influences group-by, and to the actual order of column if it is part of order-by. We make a pass on all query instances of the template to collect all features.

Example. Consider the query in Figure 5a and a configuration consisting of three indexes {I1, I2, I3} where I1= [unclustered: key = LCol1:asc], I2= [unclustered: key = OCol3,OCol2:asc], and I3=[unclus tered: key=OCol2:asc]. We create the following features:

- Selectivity of OCol2 < 10
- for column LCol1, we create the following three features: unclustered:key:scan:1:asc, unclustered:key:group-by:1:asc, unclustered:key:join:1:asc. The number 1 indicates that LCol1 is at position 1 (i.e., the leading column) among the key columns for atleast one index in the configuration.
- for column OCol3, unclustered:key:join:1:asc
- for column OCol1, unclustered:key:join:1:asc

To evaluate the accuracy, we use the geometric mean of q-error as the accuracy metric, measured as $max(\frac{est}{act}, \frac{act}{est})$ which is equivalent to minimizing the mean-squared error of the log-transformed labels [20]. As with the Index Filter, we find that tree-based ensemble models lead to sufficiently high accuracy. Since we learn one model for template, we can achieve reasonably good performance using only 5 trees with depth of 6 each, where each inference call takes less than 5 milliseconds, including the inter-process communication. This is significantly faster than an optimizer call, which can typically take 100s of milliseconds.

5.2 In-Situ Training

One of the challenges for Index Cost Models is *minimizing* the amount of training instances (i.e., optimizer calls) used for learning. While a large number of training instances is better for improving accuracy, it increases the model construction cost and the tuning time. On the other hand, training with too few examples compromises model accuracy and the quality of recommended indexes.

To address this, we propose two techniques. First, we diversify the training instances by clustering query instances as well as configurations. Second, we use an iterative approach that incrementally increases the amount of training instances, until we achieve the target accuracy. The questions that we need to answer are: (1) how to sample queries and configurations for training; and (2) how to ensure that we do not under-train or spend many more samples than needed for training. Algorithm 1 depicts how we train the cost model for each query template. We discuss the main steps below.

Clustering. To select dissimilar queries and configurations, we apply clustering. First, we cluster queries instances using k-means into $\sqrt{n}$ number of clusters(a threshold typically used in compression algorithms for SQL workloads [11, 17]) where n is the number of instances using Euclidean distance between the vector of parameter selectivities. Similarly, we cluster indexes based on their shapes. We compute the shape of the index as discussed earlier in §4.2.

Sampling of a training instance. A training instance corresponds to a (query, configuration) pair and its optimizer-estimated cost. We use the following process. We first sample a query cluster and randomly pick a query instance from the sampled cluster. We then sample the size of the configuration in a range from 1 to 4. We observe that a configuration size > 4 has negligible impact on improving the accuracy of the model and hence limit the maximum size of a configuration to 4. Given the sampled value s of configuration size, we sample s clusters of candidate indexes and randomly pick a candidate index from each cluster. We sample clusters without replacement; however, once we have covered all the clusters, we add the clusters back for re-sampling. We then make a what-if call to estimate the cost of the sampled (query, configuration) pair.

Iterative Training. We incrementally train the cost model over multiple iterations. In the first iteration, we start with a size α of training instances, and then in each of the subsequent iterations we add β more training instances. In each iteration, we use 3/4 of the sampled instances for training the model and 1/4 for validation. Given an error threshold ϵ, if the geometric mean of q-error over the validation set is less than ϵ, we say that a model is *trained*; otherwise we proceed to the next iteration. We evaluate the impact of these parameters in §6. Overall, more than 60% of Index Cost Models can be trained using less than 20% (query, candidate index) pairs. We find that setting α and β in a range of 5% to 15% of (query, candidate index) pairs results in a reasonable trade-off between model accuracy and training time (see §6), typically requiring a 2 or 3 iterations. In addition, we also set a hard-stop threshold H on the maximum size of training instance (by default ~50% of the total number of (query, candidate index) pairs for a given template) to avoid further training of cost models that do not converge quickly. For such templates (< 10% in our evaluated workloads), we fall back to the what-if calls. Nonetheless, the optimizer calls made for training such templates are not wasted: they are stored in a cache and reused during enumeration.

5.2.1 Analysis of Iterative Training. We analyze the optimality of iterative training following the similar process as discussed in [19]. Let $C_1(s)$ be the cost of generating s training examples, and let $C_2(s)$ be the cost of training an ML model with the s examples. In our problem, generating a training example requires making a what-if call to the query optimizer with the given query and index

Algorithm 1: `Training Index Cost Model for a given query template`

Input: A set of query instances belonging to the same template: Q; a set of candidate indexes: I: desired error threshold ϵ, training sample size for the first iteration: α, training sample size to add to subsequent iterations: β, hard threshold to stop training: H, total (query, candidate index) pairs for the template: T
Output: A Index Cost Model or NULL (when an accurate Index Cost Model cannot be trained)
Cluster queries Q based on selectivities ;
Cluster indexes I based on shapes ;
$S \leftarrow$ Sample α (query, configuration) pairs using the process described in §5.2;
$M \leftarrow$ Train a regression model using S ;
while *error of* $M \geq \epsilon$ **do**
  if *size of* $S > (H - \beta)$ **then**
    return *NULL*;
  end
  $N \leftarrow$ Sample β (query,configuration) pairs using the process described in §5.2;
  $S \leftarrow$ Add N training samples to previous S samples ;
  $M \leftarrow$ Re-train M using S samples ;
end
return *M;*

configuration. As a result, $C_1(s) = r \cdot C_2(s)$ where $r \gg 1$. Moreover, both $C_1(s)$ and $C_2(s)$ are proportional to the sample size s.

Let S^* be the *optimal* amount of training examples required to achieve the same prediction accuracy as that given by our iterative training algorithm.[1] The optimal cost C_{opt} is then the cost of generating these S^* examples plus the cost of training an ML model with these S^* examples. That is,

$$C_{\text{opt}} = C_1(S^*) + C_2(S^*) = (r+1) \cdot C_2(S^*).$$

Now consider the total cost C of the iterative training algorithm. The amount of training examples used in each iteration is α, $\alpha + \beta$, $\alpha + 2\beta$, etc. Let z be the smallest integer such that $\alpha + z\beta \geq S^*$. By definition we have $\alpha + (z-1)\beta < S^*$. We have the following result (see Appendix A for the proof in our full version [34]).

Theorem 1. *Assume $z \leq r$. Since $\eta = \frac{\beta}{\alpha}$, it follows that*

$$C < \left(\frac{2r+1}{r+1}\right) \cdot \left(1 + \frac{\beta}{\alpha}\right) \cdot C_{opt} < 2 \cdot \left(1 + \frac{\beta}{\alpha}\right) \cdot C_{opt}.$$

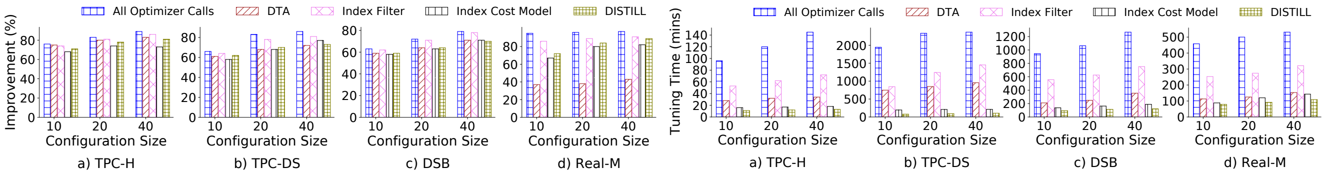
A common case is $r \gg z$, i.e., the number of iterations is much less than r given that a what-if call is very expensive compared to the amortized training cost per example. This gives $C < (1+\frac{\beta}{\alpha}) \cdot C_{\text{opt}}$, improving the optimality guarantee by a factor of 2.

5.3 Optimizations

We discuss two optimizations that are not used in DISTILL by default but can be leveraged in specific scenarios.

Using Index Filter after selecting each index in the greedy algorithm. While our proposed techniques do not leverage any properties of the greedy algorithm, and can be used with any arbitrary enumeration algorithm, we observe we can further optimize for the greedy algorithm. Specifically, by default, we use the Index Filter with the original plan to only filter syntactically-relevant indexes before performing the enumeration step. However, for an algorithm (e.g., greedy) that incrementally selects the indexes in a configuration, we can use the Index Filter to further prune candidate

[1] An implicit assumption here is that more training examples will not worsen the prediction accuracy, which is a general assumption in learning theory.



(a) Improvement **(b) Tuning time**

Figure 6: End to end evaluation of baselines and variants of DISTILL on improvement and tuning time.

indexes that may not lead to significant improvement in performance given the already selected indexes. In our experiments, we observe that this optimization further reduces the optimizer call between 10% to 15% across workloads.

Leveraging seed indexes. In some cases, index tuners can create hypothetical indexes for all relevant indexes and make one optimizer call to get the indexes (called seed indexes) selected in the most optimal plan for each query. While there may be additional indexes that may be useful to the query (but may not be selected in the most optimal plan), the selected indexes in the best plan can be used as training instances in the first iteration. We observe that doing so reduces the amount of training instances required for many templates by about 5% to 8%.

6 EXPERIMENTAL EVALUATION

Workloads. Table 3 summarizes the four workloads we use in our experiments. We use two standard benchmarks: TPC-H and TPC-DS; DSB [1], and a real customer workload Real-M. Both DSB and Real-M are more complex compared to TPC-H and TPC-DS and have skewed data distributions.

Table 3: Summary of workloads

Name	#Queries	# Templates	#Tables
TPC-H (*sf*=10)	220	22	8
TPC-DS (*sf*=10)	910	91	24
DSB [1] (*sf*=10)	520	52	24
Real-M (26GB)	160	32	474

Compared Methods. We compare the following methods: (1) a greedy algorithm as proposed in [13] and making optimizer calls for every (query, configuration) pair explored during enumeration (an ideal scenario for the maximum performance improvement for our problem setting), (2) the Database Tuning Advisor (DTA) tool [2, 6], a state-of-the-art index tuning tool that uses cost derivation using atomic configurations (described in [12]) for reducing optimizer calls. We give DTA an unlimited time budget to keep the evaluated queries and indexes similar for both DTA and DISTILL, and reduce the influence of other optimizations such as workload compression which are complementary to the techniques discussed in this paper. We consider three variants of DISTILL: (3) Index Filter for filtering syntactically-relevant indexes as discussed in §4 but making optimizer calls during enumeration as in 1), (4) Index Cost Model trained in-situ during tuning using optimizer calls and then used for costing (query, configuration) pairs during enumeration(§5), (5) DISTILL, using both (3) and (4). Unless otherwise specified, we use Random Forest (RF) with 40 trees having a maximum depth of 10 for Index Filter. To demonstrate the generalizability of Index Filter, we exclude the workload that we test from training, using only the other three workloads for training. We use a Random Forest (RF) model with 5 trees with a maximum depth of 6 for learning Index Cost Models. We provide more details on training and overhead as well as the performance of other potential ML algorithms in §6.3. The default max configuration size (k) is set to 20.

Evaluation Metrics. We use the following two metrics: (1) Improvement (%): If $C(W)$ is the original optimizer estimated cost of the workload without indexes and $C_k(W)$ is the optimizer estimated cost of the workload (when using recommended indexes), we measure improvement (%) of W as $\frac{C(W)-C_k(W)}{C(W)} \times 100\%$. (2) Time (in minutes) for tuning the workload.

6.1 End-to-End Evaluation

Figures 6a and 6b depict the impact of the compared methods on the improvement and efficiency of index tuning. The tuning times for DISTILL and Index Cost Models includes the training time of Index Cost Models. Table 4 provides more details on the number of configurations explored, the reduction in optimizer calls, as well the inference overhead. We observe that DISTILL can return recommended indexes with similar quality to those of all optimizer calls (i.e., the ideal scenario) as well as DTA in 5× to 15× less time than all optimizer calls and 2× to 12× less time than DTA. Index Filter achieves the highest improvement among our proposed techniques but takes significantly longer time compared to Index Cost Models. This is because Index Filter only removes the syntactic indexes that do not result in significant (> 5%) improvement while using optimizer calls for all explored configurations during enumeration. Index Cost Models significantly improve the tuning time by making 15× to 20× fewer optimizer calls compared to all optimizer calls. Furthermore, we observe that while there is a drop in improvement due to error in estimates, the difference is significantly small compared to DTA or all optimizer calls. When using both Index Filter and Index Cost Model, the improvement increases and the tuning time decreases, indicating the benefit of using them together. The Index Filter helps remove the spurious indexes, which in turn improves the quality of training samples for the Index Cost Models, making them converge faster with fewer training samples. Furthermore, the Index Filter also decreases the number of configurations to be evaluated by the Index Cost Models.

6.2 Impact of Number of Instances Per Template

We evaluate the impact of increasing the number of instances (while keeping the number of templates fixed). Due to space constraints, we depict the results using improvement per unit time. As the number of instances increases, the performance of Index Cost Model and the DISTILL improves, while for the baselines and the Index Filter, the relative performance is less variant to the number of instances. When the number of instance is 1, i.e., each query with a different query template, training Index Cost Models consumes a substantial fraction of the tuning time, and hence the difference between Index Cost Models and DTA is not significant. However, by only increasing the number of query instances to only 3, we see that Index Cost Models perform much better.

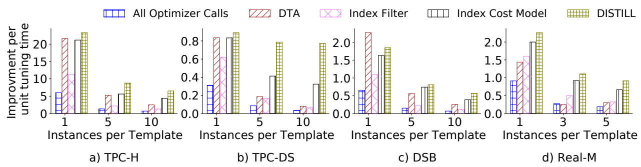


Figure 7: Impact of number of instances per template

Table 4: Analysis of Index Filter and Index Cost Models

	TPC-H	TPC-DS	DSB	Real-M
Total (query, syntactic index) pairs	3284	11351	8832	7350
Total configurations explored by all optimizer calls	18534	87334	45294	23568
Total (query, syntactic index) pairs after Index Filter	1832	6842	4042	3562
Total configurations explored by DISTILL	8231	34204	20565	9445
Total optimizer calls made by DISTILL	964	4021	2459	1754
Total Index Cost Models	20	86	49	29
Average training instance size for each Index Cost Models	18	25	32	41
Average training time for each Index Cost Models (excluding optimizer calls)	4s	5s	7s	8s
Average inference time for Index Filters	8 ms	10ms	10ms	10ms
Average inference time for Index Cost Models	5ms	5ms	5m	5ms

6.3 Effectiveness of ML algorithms

6.3.1 Index Filter. We first compare the effectiveness of ML algorithms as Index Filter. We test the model over one workload at a time, while using the other three workloads for training. We considered the following regression models: Logistic regression (LR), three tree-based ensemble models: Light GBM (LGBM), XGBoost, and Random Forest (RF) with hyperparameters tuned using FLAML [36]. The optimal number of trees varies between 30 and 60 with depth of trees between 5 to 20. In general, we observe that RF with 40 trees and a depth of 10 results in best performance for most of the workloads. We also consider a feed-forward fully-connected neural network implemented using MLPRegressor [3]. We manually tuned the parameters and found that 3 hidden layers with ReLU as the activation function, adam as optimizer using a maximum of 300 epochs results in the best performance. For all models, we use log-transformed labels and mean square error as the loss function. We use 0.05 as the threshold for identifying spurious indexes, and the F1 score for evaluation which captures both the precision and recall of identifying spurious indexes.

As depicted in Figure 8a, we observe that most of the tree-based ensemble models perform significantly better than the default approach of generating syntactically-relevant indexes as in DTA. On the other hand, we find that LR performs worse among all the ML techniques. On average, we observe that tree-based models have a precision of about 0.75 and a recall of 0.85 across all the workloads. Furthermore, the worst precision and recall occur over the DSB benchmark (a more complex workload), with the highest values among all models as 0.71 and 0.79, whereas the best precision and recall happen for the TPC-H workload with the values of 0.78 and 0.95 respectively. On average, we see that we are able to reduce the false positive rate by 70% with a low false negative rate between 4% and 14%, and an overall reduction of about 42% of the syntactic indexes from enumeration using the tree-based ensemble models.

Regression vs. Classification. We also consider learning Index Filter as a classification task where we label all training instances where improvement fraction > .05 as 1 and those with improvement fraction < .05 as 0. We depict only the RF- and MLP-based classifiers in Figure 8a as performance of other classifiers are worse. We find

Table 5: Sensitivity of features for Index Filter measured using F-1 score while turning off each category of features

Workload	w/o utility	w/o shape	w/o operators	w/o optimizer behaviour
TPC-H	.60	.69	.68	.78
TPC-DS	.30	.50	.53	.74
DSB	.35	.40	.59	.71
Real-M	.49	.65	.57	.77

learning the model as classifier results in lower F1 score compared to the regression models. This is because the regression model is able to better model how changes in feature values impact the cost, while with the binary labeling, we lose this information.

Overhead. On average, it takes between 80 and 130 hours to generate the labelled training data (requiring optimizer calls) using the databases and workloads listed in Table 3, consisting of a maximum of 170k (query, configuration) pairs and their optimizer estimated costs. The training takes less than 5 minutes for the tree-based ensemble models and about 20 minutes for neural network models using CPU. The inference time for tree-based ensemble models is about 10 ms (significantly faster than optimizer calls) while for MLP-based models it is in order of 100 of milliseconds.

6.3.2 Index Cost Model. We considered the same ML algorithms as above. For the neural network model, we use the same setting as for the Index Filter except that we use only 100 epochs for training and 2 hidden layers instead of 3 as adding more epochs and layers results in a significant increase in tuning time without much improvement in accuracy. For this experiment, we perform a 5-fold cross-validation with a training size set to 25% of the (query, candidate index) pairs. Again, we observe that tree-based models perform significantly better than the LR model. On average, we see that the median q-error of tree-based models across all workloads is around 1.18. For tree-based models, we can achieve reasonably good performance using between 4 to 10 trees with depth of trees ranging between 3 to 8.

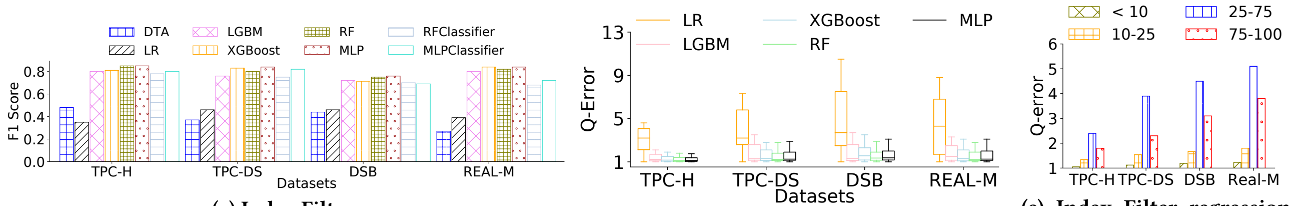
Overhead. The training of Index Cost Models varies between 20 seconds to 100 seconds depending on the number of optimizer calls. On average, about 35 optimizer calls are required to train a model for each template. Training of ML model given the labelled instances typically takes less than 5 seconds. Each inference call takes less than 5 milliseconds, including the inter-process communication. This is significantly faster than an optimizer call, which take several 100s of milliseconds.

6.4 Sensitivity of Features in Index Filter

The Index Filter uses four types of signals as features: (1) utility, (2) shape, (3) operators, and (4) optimizer behaviour captured via table and index details. We turned off each feature one at a time, and tested the F1 score of the Index Filter. For this experiment, we use RF as our choice of ML model. Table 5 depicts the results. We see that utility has the maximum influence on the F1 score, the removal of which results in the maximum decrease in the F1 score. This is followed by shape-based and operator-based features. Finally, we see that the optimizer behaviour captured via table and index details have the minimum impact on the F1 score.

6.5 Index Cost Model vs Alternative Techniques

6.5.1 Index Cost Model vs cost derivation using atomic configurations [12]. We compare the two approaches using the same set of candidate input indexes. Table 6 depicts the results for cost-derivation. Note that cost derivation works only with the greedy algorithm while the Index Cost Model is agnostic of the enumeration



(a) Index Filter

(b) Index Cost Model

(c) Index Filter regression model for costing

Figure 8: a) and b) evaluate the effectiveness of different ML algorithms for Index Filter and Index Cost Model. In c), we evaluate the effectiveness of using Index Filter regression model for costing. The (query, configuration) pairs are grouped based on actual improvements, e.g., the bar < 10 represents a group of (query, configuration) pairs where actual improvement is $< 10\%$.

Table 6: Cost derivation using atomic configurations

Name	#Optimizer Calls	Improvement	Tuning Time (mins)
TPC-H	1739	75%	31
TPC-DS	1324	63%	748
DSB [1]	8394	63%	395
Real-M	5272	54%	184

algorithm. Despite this, we find cost derivation making between 2× to 6× more optimizer calls and taking proportionally more time compared to Index Cost Models (see Figure 6a and Figure 6b) while giving significantly worse improvement over complex workloads such a Real-M. One might see that cost derivation takes much more time than DTA. This is because DTA in addition to cost derivation also leverages other optimizations such as index merging, table-subset selections [6] which reduces the search space.

6.5.2 Using Index Filter regression model for costing. To understand whether we can use the Index Filter regression model for costing, we grouped (query, configuration) pairs into four groups based on their actual improvements. As depicted in Figure 8c, Index Filter is effective at identifying configurations with small improvements in performance (e.g., < 5% improvement) as depicted via a low q-error. However, for configurations which result in significant improvement, Index Filter suffers with large q-errors than the workload-specific Index Cost Models.

6.6 Evaluation of Training of Index Cost Model

6.6.1 Impact of training size. Figure 9a depicts the % of query templates for which we use learn cost models as we vary the fraction of the total (query, candidate index) pairs for training. A model is trained when the geometric mean of q-error on the validation set is < 1.20. On average, with training size around 0.20 fraction, we are able to train about 65% of the models across all workloads. With an additional 20% of training instances, we are able to train over 85% of the models, after which we see that adding more training samples has marginal impact on reducing the error of the models.

6.6.2 Impact of clustering. Figures 9a and 9b depict the impact of clustering of queries as well as indexes on the training size required to train models. As in the above experiment, a model is trained if the geometric mean of q-error over the validation set is below 1.20. As depicted, we are able to train a large number of models with fewer training instances by clustering. For instances, with .20 training fraction, we are able to train about 65% models (on average across workloads) with clustering; on the the other hand, we can train only 40% of models without clustering.

6.6.3 Varying threshold on maximum error (ϵ). Figure 9c depicts the impact of error threshold (measured using geometric mean of q-error) used for training the cost model on the improvement and the end-to-end tuning time captured via improvement per unit tuning time. We observe that when we set the error threshold to close to 1.0, the improvement is very close to making all optimizer calls; however, the tuning time is also high — about 2×-3× more than the tuning time it takes for the default value 1.20. As we increase the error threshold from 1.05 to 1.50, we see a drastic reduction (3×-5×) in tuning time with an average reduction of 8% in the improvement across all workloads. On further relaxing the error threshold, we see a faster reduction in improvement due to less accurate cost models. Overall, a threshold between 1.20 and 1.50 results in much faster tuning with small reduction in quality of indexes, compared to making all optimizer calls.

6.6.4 Varying α, β. Figure 9d and 9e depict the impact on tuning time as we vary the threshold α (initial training size) and β (additional training size) added per iteration for incrementally training cost models. On average, as depicted in Figure 9a, we see that training size ranging between 30% to 50% of the total (query-candidate index) pairs is sufficient in training most of the cost models. The choices of α and β determine the number of times we need to train the models. Overall, we see that different values of α and β in a range of 5% to 15% give similar results in tuning time with the number of iterations ranging between 2 to 5.

6.7 Impact of Optimizations

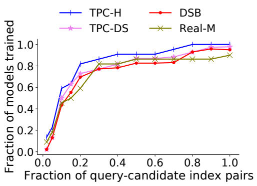
Figure 10 depicts the impact of optimizations proposed in §5.3.

Pruning after selecting each index during greedy selection. After selecting each index during greed enumeration, we use the Index Filter to prune out candidate indexes that are less likely to result in improvement of cost. We find that doing so results in between 10% and 15% decrease in tuning time across workloads with a small reduction (< 3%) in quality.

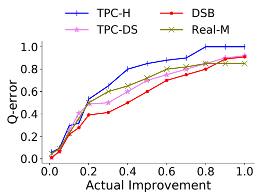
Using seed indexes for training. During the first iteration while training Index Cost Models, we add the seed indexes as training samples. As depicted in Figure 10b, we observe that doing so results in faster convergence of cost models as compared to Figure 9a. For instance, we observe that we can learn about about 18% of Index Cost Models on TPC-DS workload when using seed indexes compared to 10% of model with the same fraction of training samples but not using seed indexes.

7 RELATED WORK

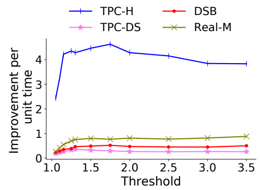
Prior work have proposed techniques [12, 22, 28] that cache physical plans across optimizer calls and reason about the reuse of costs of sub-expressions across queries using rules or external cost models. We observe that these techniques require many more optimizer calls than used for training Index Cost Models (see §6.5). Furthermore, unlike DISTILL, these techniques cannot prune spurious indexes, and require at least one optimizer call to identify such



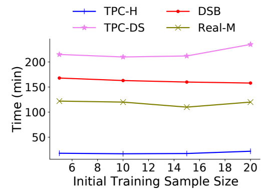
(a) Clustered Sampling



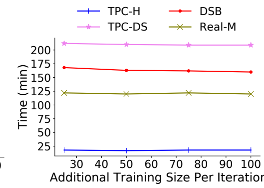
(b) Unclustered Sampling



(c) Impact of error threshold

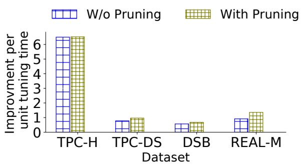


(d) Impact of α

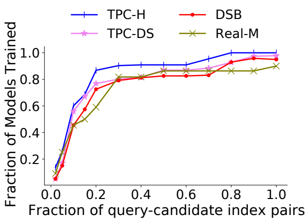


(e) Impact of β

Figure 9: Sensitivity of parameters used in iterative training



(a) Pruning after selecting each index using greedy algorithm



(b) Impact of seed indexes

Figure 10: Impact of Optimizations

indexes. Finally, the reuse of costs heavily depends on the order of enumeration of indexes, thereby requiring changes to the tuning algorithm to be effective. There have also been techniques that extend the traditional candidate selection through merging candidate indexes [15], thereby reducing the number of configurations to explore. [8, 9] compute the bounds on costs of queries based on query optimization of past configurations, which can be used for pruning optimizer calls. Unlike our work, these techniques require changes either to the index tuning algorithm or to the optimizer.

Workload compression techniques [12, 17] have been proposed for reducing the size of the input workload that are complementary to the techniques discussed in this work. These techniques typically operate independent of the index tuning tool and are therefore unaware of index tuning constraints, such as storage budget and size of configurations that play an important factor in selecting indexes. Furthermore, these techniques either select or discard a query completely; however, parts of the query may still be useful.

There has been work on applying *reinforcement learning* techniques for index tuning [7, 27, 29–31] that primarily focus on search enumeration and target online tuning scenarios where queries arrive in a sequence. In contrast, we focus on the classical offline index tuning problem where we can access the entire workload at once. Ding et al. [18] have proposed learning a classifier for reducing query performance regressions due to erroneous optimizer cost models. Specifically, the proposed featurization and ML techniques focus on a different setting, i.e., comparing two physical plans (by making optimizer calls) to predict regression and hence cannot be adapted for estimating costs for (query, configuration) pairs.

Finally, the work on parametric query optimization (PQO) [22–24] studies the change in the optimal plan for a query under changing parameters, such as predicate selectivity. The Index Cost Models in our work have the same goal except that we consider both parameters and index configurations, and develop optimizations based on how indexes are used within a query plan to reduce the number of training instances.

8 DISCUSSION

In this work, we assume that the physical plan for each query in the workload is provided as input. We observe that most DBMSs expose functionality to collect historical workload information including query execution plans, e.g., Query Store [4] in Microsoft SQL Server. Such information can be leveraged by DISTILL for analyzing queries without making optimizer calls. However, in cases where such logs are not available, DISTILL needs to make an optimizer call for each query in the workload to get its execution plan. For large input workloads, the overheads of making such calls may be significant.

Additionally, index advisors (e.g., see DTA [6]) support tuning with a time-budget, where queries from the input workload are consumed and tuned incrementally. While the Index Filter can seamlessly operate in such a setting, the accuracy of the Index Cost Models may suffer when a small (e.g., < 3 over our evaluated workloads) subset of instances for a query template are available for in-situ training. One option to address this could be to select all instances of a query template together when consuming the workload incrementally.

There have been complementary techniques on workload compression [33] that identify a subset of queries (thereby reducing the search space) which when tuned result in indexes that improve the performance of the entire workload. In contrast, in this work, we use efficient techniques for estimating the cost of (query, configuration) pairs without reducing the search space. An interesting next step could be to combine both the techniques to further reduce the tuning time without affecting the performance of the workload.

9 CONCLUSION

In this paper, we described the scalability challenges with index tuning tools and discuss a number of opportunities for improvement. We discussed how we can leverage machine learning techniques to reduce the amount of optimizer calls without making any changes to the index tuning algorithm or to the query optimizer. First, we presented how we can learn a workload-agnostic model that captures patterns over query structure, statistics, and index structure to identify indexes that do not lead to plan changes. This model can be used to remove a large number of syntactically-relevant but spurious index candidates. Next, we showed how we can learn an index-specific cost models on-the-fly during the tuning process and use them to *predict* the costs of many (query, configuration) pairs during enumeration, thereby avoiding optimizer calls. Our results show that our proposed techniques result in significant improvement in tuning time while recommending indexes with similar quality as state-of-the-art approaches.

REFERENCES

[1] Apr 01, 2022. DSB Benchmark. https://github.com/microsoft/dspp-benchmark.
[2] Apr 01, 2022. DTA utility. https://docs.microsoft.com/en-us/sql/tools/dta/dta-utility?view=sql-server-ver15.
[3] Apr 01, 2022. MLPRegressor. https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html.
[4] Apr 01, 2022. Query Store. https://docs.microsoft.com/en-us/sql/relational-databases/performance/monitoring-performance-by-using-the-query-store?view=sql-server-ver15.
[5] Apr 01, 2022. Statistics. https://docs.microsoft.com/en-us/sql/relational-databases/statistics/statistics?view=sql-server-ver15.
[6] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollar, Arun Marathe, Vivek Narasayya, and Manoj Syamala. 2005. Database tuning advisor for microsoft sql server 2005. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. 930–932.
[7] Debabrota Basu, Qian Lin, Weidong Chen, Hoang Tam Vo, Zihong Yuan, Pierre Senellart, and Stéphane Bressan. 2015. Cost-Model Oblivious Database Tuning with Reinforcement Learning. In *DEXA*. 253–268.
[8] Nicolas Bruno and Surajit Chaudhuri. 2005. Automatic Physical Database Tuning: A Relaxation-based Approach. In *SIGMOD*. 227–238.
[9] Nicolas Bruno and Surajit Chaudhuri. 2006. To tune or not to tune? A Lightweight Physical Design Alerter. In *Proceedings of the 32nd international conference on Very large data bases*. Citeseer, 499–510.
[10] Nicolas Bruno and Surajit Chaudhuri. 2007. Physical design refinement: The 'merge-reduce' approach. *ACM Trans. Database Syst.* 32, 4 (2007), 28.
[11] Surajit Chaudhuri, Mayur Datar, and Vivek R. Narasayya. 2004. Index Selection for Databases: A Hardness Study and a Principled Heuristic Solution. *IEEE Trans. Knowl. Data Eng.* 16, 11 (2004), 1313–1323.
[12] Surajit Chaudhuri, Ashish Kumar Gupta, and Vivek R. Narasayya. 2002. Compressing SQL workloads. In *SIGMOD*. 488–499.
[13] Surajit Chaudhuri and Vivek R. Narasayya. 1997. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB*. 146–155.
[14] Surajit Chaudhuri and Vivek R. Narasayya. 1998. AutoAdmin 'What-if' Index Analysis Utility. In *SIGMOD*. 367–378.
[15] Surajit Chaudhuri and Vivek R. Narasayya. 1999. Index Merging. In *ICDE*.
[16] Sudipto Das, Miroslav Grbic, Igor Ilic, Isidora Jovandic, Andrija Jovanovic, Vivek R. Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu, and Surajit Chaudhuri. 2019. Automatically Indexing Millions of Databases in Microsoft Azure SQL Database. In *SIGMOD*. 666–679.
[17] Shaleen Deep, Anja Gruenheid, Paraschos Koutris, Jeffrey F. Naughton, and Stratis Viglas. 2020. Comprehensive and Efficient Workload Compression. *Proc. VLDB Endow.* 14, 3 (2020), 418–430.
[18] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. 2019. AI Meets AI: Leveraging Query Executions to Improve Index Recommendations. In *SIGMOD*. 1241–1258.
[19] Anshuman Dutt, Chi Wang, Vivek Narasayya, and Surajit Chaudhuri. 2020. Efficiently approximating selectivity functions using low overhead regression models. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2215–2228.
[20] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. 2019. Selectivity estimation for range predicates using lightweight models. *Proceedings of the VLDB Endowment* 12, 9 (2019), 1044–1057.
[21] S. J. Finkelstein, M. Schkolnick, and P. Tiberio. 1988. Physical Database Design for Relational Databases. *ACM Trans. Database Syst.* 13, 1 (1988).
[22] Antara Ghosh, Jignashu Parikh, Vibhuti S Sengar, and Jayant R Haritsa. 2002. Plan selection based on query clustering. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 179–190.
[23] Naveen Reddy Jayant R Haritsa. 2005. Analyzing plan diagrams of database query optimizers. In *Proceedings of the 31st International Conference on Very Large Data Bases. VLDB Endowment*. 1228–1239.
[24] Arvind Hulgeri and S Sudarshan. 2002. Parametric query optimization for linear and piecewise linear cost functions. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 167–178.
[25] Alekh Jindal, Hiren Patel, Abhishek Roy, Shi Qiao, Zhicheng Yin, Rathijit Sen, and Subru Krishnan. 2019. Peregrine: Workload optimization for cloud query engines. In *Proceedings of the ACM Symposium on Cloud Computing*. 416–427.
[26] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. 2020. Magic mirror in my hand, which is the best in the land? An Experimental Evaluation of Index Selection Algorithms. *Proc. VLDB Endow.* 13, 11 (2020), 2382–2395.
[27] Hai Lan, Zhifeng Bao, and Yuwei Peng. 2020. An Index Advisor Using Deep Reinforcement Learning. In *CIKM*. 2105–2108.
[28] Stratos Papadomanolakis, Debabrata Dash, and Anastassia Ailamaki. 2007. Efficient Use of the Query Optimizer for Automated Database Design. ACM.
[29] Malinga Perera, Bastian Oetomo, Benjamin I. P. Rubinstein, and Renata Borovica-Gajic. 2020. DBA bandits: Self-driving index tuning under ad-hoc, analytical workloads with safety guarantees. *CoRR* abs/2010.09208 (2020).
[30] Zahra Sadri, Le Gruenwald, and Eleazar Leal. 2020. Online Index Selection Using Deep Reinforcement Learning for a Cluster Database. In *ICDE Workshops 2020*. 158–161.
[31] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. 2018. The Case for Automatic Database Administration using Deep Reinforcement Learning. *CoRR* abs/1801.05643 (2018).
[32] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. 2020. Cost models for big data query processing: Learning, retrofitting, and our findings. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 99–113.
[33] Tarique Siddiqui, Saehan Jo, Wentao Wu, Chi Wang, Vivek Narasayya, and Surajit Chaudhuri. 2022. ISUM: Efficiently Compressing Large and Complex Workloads for Scalable Index Tuning. In *Proceedings of the 2022 International Conference on Management of Data*. 660–673.
[34] Tarique Siddiqui, Wentao Wu, Vivek Narasayya, and Surajit Chaudhuri. 2022. DISTILL: Low-Overhead Data-Driven Techniques for Filtering and Costing Indexes for Scalable Index Tuning (MSR Technical Report. To be published at VLDB 2022.). https://www.microsoft.com/en-us/research/publication/distill/.
[35] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy M. Lohman, and Alan Skelley. 2000. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *ICDE*. 101–110.
[36] Chi Wang, Qingyun Wu, Markus Weimer, and Erkang Zhu. 2021. FLAML: a fast and lightweight AutoML Library. *Proceedings of Machine Learning and Systems* 3 (2021), 434–447.
[37] Wentao Wu, Chi Wang, Tarique Siddiqui, Junxiong Wang, Vivek Narasayya, Surajit Chaudhuri, and Philip A Bernstein. 2022. Budget-aware Index Tuning with Reinforcement Learning. In *Proceedings of the 2022 International Conference on Management of Data*. 1528–1541.

A ANALYSIS OF ITERATIVE TRAINING

Let $C_1(s)$ be the cost of generating s training examples, and let $C_2(s)$ be the cost of training an ML model with the s examples. In our problem, generating a training example requires making a what-if call to the query optimizer with the given query and index configuration. As a result, $C_1(s) = r \cdot C_2(s)$ where $r \gg 1$. Moreover, both $C_1(s)$ and $C_2(s)$ are proportional to the sample size s.

Let S^* be the *optimal* amount of training examples required to achieve the same prediction accuracy as that given by our iterative training algorithm.[2] The optimal cost C_{opt} is then the cost of generating these S^* examples plus the cost of training an ML model with these S^* examples. That is,

$$C_{\text{opt}} = C_1(S^*) + C_2(S^*) = (r+1) \cdot C_2(S^*).$$

Now consider the total cost C of the iterative training algorithm. The amount of training examples used in each iteration is A, $A + B$, $A + 2B$, etc. Let z be the smallest integer such that $A + zB \geq S^*$. By definition we have

$$A + (z-1)B < S^*. \tag{1}$$

For ease of exposition, we introduce $\eta = \frac{B}{A}$ and thus $B = \eta A$. As a result, we have

$$\Big(1 + (z-1)\eta\Big) \cdot A < S^*,$$

which yields

$$A < \frac{S^*}{1 + (z-1)\eta}. \tag{2}$$

We can express C as follows:

$$C = C_1(A + zB) + \sum_{i=0}^{z} C_2(A + iB). \tag{3}$$

Let $c_1 = C_1(A + zB)$ and $c_2 = \sum_{i=0}^{z} C_2(A + iB)$. By Equation 1,

$$\begin{aligned} c_1 &= C_1\Big(A + (z-1)B\Big) + C_1(B) \\ &< C_1(S^*) + C_1(\eta A). \end{aligned}$$

By Equation 2, it follows that

$$\begin{aligned} c_1 &< C_1(S^*) + \frac{\eta}{1 + (z-1)\eta} \cdot C_1(S^*) \\ &= \frac{1 + z\eta}{1 + (z-1)\eta} \cdot C_1(S^*). \end{aligned} \tag{4}$$

On the other hand, we have

$$\begin{aligned} c_2 &= C_2\Big(\sum_{i=0}^{z} (A + iB)\Big) \\ &= C_2\Big((z+1)A + \frac{z(z+1)}{2} \cdot B\Big) \\ &= (z+1) \cdot C_2(A + \frac{z}{2} \cdot B) \\ &= (z+1) \cdot C_2\Big((1 + \eta \cdot \frac{z}{2}) \cdot A\Big). \end{aligned}$$

Again, by Equation 2, it follows that

$$c_2 < (z+1) \cdot \Big(\frac{1 + \eta \cdot \frac{z}{2}}{1 + (z-1)\eta}\Big) \cdot C_2(S^*). \tag{5}$$

Combining Equations 3, 4, and 5, we have

$$C = c_1 + c_2 < \frac{(1 + z\eta) \cdot C_1(S^*) + (z+1) \cdot \Big(1 + \eta \cdot \frac{z}{2}\Big) \cdot C_2(S^*)}{1 + (z-1)\eta}.$$

[2] An implicit assumption here is that more training examples will not worsen the prediction accuracy, which is a general assumption in learning theory.

Since $C_1(S^*) = r \cdot C_2(S^*)$ and $C_{\text{opt}} = (r+1) \cdot C_2(S^*)$, it follows that

$$\begin{aligned} C &< \frac{r \cdot \Big(1 + z\eta\Big) + (z+1) \cdot \Big(1 + \eta \cdot \frac{z}{2}\Big)}{1 + (z-1)\eta} \cdot C_2(S^*) \\ &= \frac{r \cdot \Big(1 + z\eta\Big) + (z+1) \cdot \Big(1 + \eta \cdot \frac{z}{2}\Big)}{(r+1) \cdot \Big(1 + (z-1)\eta\Big)} \cdot C_{\text{opt}}. \end{aligned} \tag{6}$$

Since $\eta \cdot \frac{z}{2} < \eta \cdot z$, we have

$$1 + \eta \cdot \frac{z}{2} < 1 + \eta \cdot z = 1 + (z-1)\eta + \eta.$$

By Equation 6, it follows that

$$\begin{aligned} C &< \frac{r \cdot \Big(1 + z\eta\Big) + (z+1) \cdot \Big(1 + (z-1)\eta + \eta\Big)}{(r+1) \cdot \Big(1 + (z-1)\eta\Big)} \cdot C_{\text{opt}} \\ &= \frac{r \cdot \Big(1 + z\eta\Big) + (z+1) \cdot \eta + (z+1) \cdot \Big(1 + (z-1)\eta\Big)}{(r+1) \cdot \Big(1 + (z-1)\eta\Big)} \cdot C_{\text{opt}} \\ &= \left(\frac{r \cdot \Big(1 + z\eta\Big) + (z+1) \cdot \eta}{(r+1) \cdot \Big(1 + (z-1)\eta\Big)} + \frac{z+1}{r+1}\right) \cdot C_{\text{opt}}. \end{aligned}$$

Applying $1 + \eta \cdot z = 1 + (z-1)\eta + \eta$ again, we obtain

$$\begin{aligned} C &< \left(\frac{r \cdot \Big(1 + (z-1)\eta\Big) + (r+z+1) \cdot \eta}{(r+1) \cdot \Big(1 + (z-1)\eta\Big)} + \frac{z+1}{r+1}\right) \cdot C_{\text{opt}} \\ &= \left(\frac{r}{r+1} + \frac{(r+z+1) \cdot \eta}{(r+1) \cdot \Big(1 + (z-1)\eta\Big)} + \frac{z+1}{r+1}\right) \cdot C_{\text{opt}} \\ &= \left(\frac{r+z+1}{r+1} + \frac{(r+z+1) \cdot \eta}{(r+1) \cdot \Big(1 + (z-1)\eta\Big)}\right) \cdot C_{\text{opt}} \\ &= \left(\frac{r+z+1}{r+1}\right) \cdot \left(1 + \frac{\eta}{1 + (z-1)\eta}\right) \cdot C_{\text{opt}}. \end{aligned}$$

Assuming $z \geq 1$ (i.e., we need at least one iteration), it follows that

$$C < \left(\frac{r+z+1}{r+1}\right) \cdot (1 + \eta) \cdot C_{\text{opt}}. \tag{7}$$

Based on Equation 7, we have the following observations.

Theorem 2. *Assume $z \leq r$. Since $\eta = \frac{B}{A} = \frac{\beta}{\alpha}$, it follows that*

$$C < \left(\frac{2r+1}{r+1}\right) \cdot \left(1 + \frac{\beta}{\alpha}\right) \cdot C_{opt} < 2 \cdot \left(1 + \frac{\beta}{\alpha}\right) \cdot C_{opt}.$$

On the other hand, if $r \gg z$, which is the common case in practice (i.e., the number of iterations is much less than r given that a what-if call is very expensive compared to the amortized training cost per example), then we have $C < (1 + \frac{\beta}{\alpha}) \cdot C_{\text{opt}}$, which improves the optimality guarantee by a factor of 2.