AI Meets AI: Leveraging Query Executions to Improve Index Recommendations

Bailu Ding[§] Sudipto Das[§] Ryan Marcus[†] Wentao Wu[§]
Surajit Chaudhuri[§] Vivek R. Narasayya[§*]

[§]Microsoft Research [†]Brandeis University

ABSTRACT

State-of-the-art index tuners rely on query optimizer's cost estimates to search for the index configuration with the largest estimated execution cost improvement‘. Due to well-known limitations in optimizer's estimates, in a significant fraction of cases, an index estimated to improve a query's execution cost, e.g., CPU time, makes that worse when implemented. Such errors are a major impediment for automated indexing in production systems.

We observe that comparing the execution cost of two plans of the same query corresponding to different index configurations is a key step during index tuning. Instead of using optimizer's estimates for such comparison, our key insight is that formulating it as a classification task in machine learning results in significantly higher accuracy. We present a study of the design space for this classification problem. We further show how to integrate this classifier into the state-of-the-art index tuners with minimal modifications, i.e., how *artificial intelligence (AI) can benefit automated indexing (AI).* Our evaluation using industry-standard benchmarks and a large number of real customer workloads demonstrates up to 5× reduction in the errors in identifying the cheaper plan in a pair, which eliminates almost all query execution cost regressions when the model is used in index tuning.

[*]Ryan Marcus performed the work while at Microsoft Research.

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3324957

CCS CONCEPTS

• **Information systems** → **Autonomous database administration**; *Data layout*; • **Computer systems organization** → **Cloud computing**;

KEYWORDS

Automated indexing; autonomous database management; performance tuning; relational database-as-a-service.

ACM Reference Format:
Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, Vivek R. Narasayya. 2019. AI Meets AI: Leveraging Query Executions to Improve Index Recommendations. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3299869.3324957

1 INTRODUCTION

Motivation. Selecting an appropriate set of indexes for a given workload can result in significant reductions in query execution cost, e.g., CPU time. Thus, recommending indexes for databases has been an active research area for several decades [2, 12, 17–19, 23, 25, 30, 56, 65, 72]. Being able to *fully automate index recommendation and implementation* is a significant value-add. One key requirement of automated index implementation for production systems is that creating or dropping indexes does not cause significant **query performance regressions**. Such regressions, where a query's execution cost increases after changing the indexes, is a major impediment to fully-automated indexing [24, 52] as users desire to enforce a **no query regression constraint**.

State-of-the-art industrial-strength index tuning systems [2, 23, 72] rely on query optimizer's cost estimates to recommend indexes with the most estimated improvement. When a tuner searches for alternative index configurations, it needs to *compare execution cost* of different plans for the same query that correspond to different configurations. The tuner can enforce the no regression constraint using the optimizer's estimates. However, due to well-known limitations in optimizer's estimates, such as errors in cardinality estimation

or cost model [47, 50, 68], using the optimizer's estimates to enforce the constraint can result in significant errors.

Cost estimation errors. To comprehend the challenge even in the simplified task of comparing the cost of two plans, consider a plan pair $\langle \mathcal{P}_1, \mathcal{P}_2 \rangle$ for the same query. Figure 1 reports the ratio $\frac{Cost(\mathcal{P}_2)}{Cost(\mathcal{P}_1)}$ computed with CPU time (clipped between 0.01 and 100) on the y-axis and the optimizer's cost estimate on the x-axis. We only consider a small set of randomly-selected pairs from TPC benchmarks [63, 64] and several real customer workloads where the *optimizer estimates $\mathcal{P}_2$ to be cheaper than $\mathcal{P}_1$*.

Any point whose y value is < 1 (blue circles) is an improvement in execution cost, while whose value is > 1 is a regression (orange ×). Note that in $\sim$ 20–30% cases, an estimated improvement is a regression, with several instances where a plan estimated to be 2–10× cheaper ends up with 2× or more regression. Hence, using the optimizer's cost estimates to enforce the no regression constraint still results in significant regressions compared to the existing configuration [10, 26, 28].

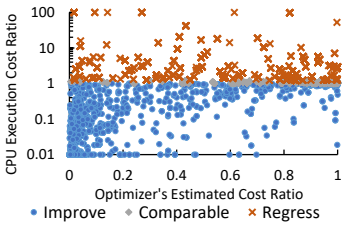


Figure 1: The ratio of CPU execution cost vs. that of optimizer's estimated cost.

A cloud database provider, such as Microsoft Azure SQL Database, can observe aggregated (and anonymized) execution statistics over millions of databases. When databases are auto-indexed, we collect execution statistics for several plans of the same query across different index configurations. Given this huge execution data repository, we ask: *How can an index tuner effectively leverage this data to improve its index recommendation quality in terms of execution cost?*

Our Approach: We present a design and implementation for state-of-the-art index tuners [2, 19, 72] to effectively leverage machine learning (ML) techniques to improve index recommendation quality, i.e., leverage *artificial intelligence (AI) to improve automated indexing (AI)*. It is common to use execution history to train an ML model to predict execution cost [5, 27, 31, 49] by formulating the problem as a regression task.[1] The index tuner can then use the ML model's predicted cost instead of the query optimizer's estimated cost. However, due to the huge diversity of queries, data distributions, physical operator types, and index types, this regression problem is challenging. Not surprising, all the existing approaches for cost prediction report significant errors when compared to true execution cost [5, 49, 68].

Our **first key insight** is that comparing execution cost of two plans can be formulated as a classification task in ML. Training a classifier to decide which plan has cheaper execution cost among a pair of plans results in higher accuracy in contrast to using a (learned or analytical) cost model to compare cost. While we present empirical evidence backing this insight in Section 7.5, an intuitive explanation lies in the error metric that each ML task minimizes. The ML regression task minimizes the prediction error for each plan. If accurate, such a model is much more general and versatile compared to the classification approach. However, such models have significant prediction errors in practice [5, 49, 68], which could translate to significant errors in cost comparison. Similarly, a regression model that learns ratio of execution cost on pairs of plans minimizes the error for predicting ratios instead of that for predicting comparisons. In contrast, the classifier learns on pairs of plans and can *directly minimize the error metric that corresponds to comparison errors*, which results in significantly higher accuracy in practice.

We present the design space for this classification task: (*a*) how to featurize query plans (i.e., trees of operators) into vectors (Section 3); and (*b*) how to learn such a classifier (Section 4). To leverage execution data collected from millions of databases in the cloud, such as in Azure SQL Database, we learn an *offline* model across the databases (Section 4.1). An important assumption for ML models to perform well is that the train/test data have similar distributions. As we discuss in Section 4.2, such an assumption often does not hold across queries within a database. The huge diversity of schema, data distributions, and query workloads that a cloud database service hosts makes it even more challenging. We present a lightweight approach to *adapt* the *offline* cross-database model to individual databases and workloads with very little additional execution data from the new distributions (Section 4.3) that significantly improves accuracy over the offline model. This adaptation addresses an important challenge neglected by most applications of ML techniques. We also present alternative modeling techniques in Section 6.

Existing index tuners use the optimizer's *cost estimates* to be "in-sync" with the query optimizer [17, 25, 30, 65], i.e., ensure that the optimizer will use the indexes if implemented. Our **second key insight** is that being "in-sync" with the optimizer only requires the tuner to use the *plan* the optimizer will pick. We implement our technique in an prototype index tuner for Azure SQL Database that builds on the concepts described in literature [2, 17, 19, 65, 72]. We extend the tuner to use the optimizer's "what-if" API [18] to obtain plans for hypothetical configurations and leverage the classifier to predict whether a query's plan in a hypothetical configuration is

[1] The machine learning regression task should not be confused with query performance regression.

cheaper compared to that in the existing configuration (Section 5). For a workload (i.e., a set of queries), we leverage the classifier to enforce (with a high probability) the *constraint* that no single query regresses and recommend a configuration that minimizes the estimated cost of the workload.

In Section 7, we present a thorough evaluation of the proposed techniques using a large collection of industry-standard TPC benchmarks and real customer workloads. Our experiments show that the classification-based approaches have significantly higher F1 score in identifying the more expensive plan from a plan pair, compared to several other regression-based approaches as well as the query optimizer. Our techniques reduce the errors in the plan pair comparison by up to 5×, especially when train/test distributions are similar. We observe that for this classification task, a Random Forest trains over millions of data points in tens of minutes on commodity servers, results in model size of tens of megabytes, and has high prediction accuracy. An index tuner augmented with the classifier has very few regressions and has improvement comparable with the original tuner. The overhead of using the model during tuning is negligible.

In summary, our work makes the following contributions:

- We present our key insight that training a classifier to compare the cost of two plans (a common task in an index tuner) can be more accurate than learning to predict the cost for a plan and then comparing the cost.
- We present a technique to featurize query plans into vectors and present a thorough study of model alternatives.
- We present a lightweight approach to *adapt* the models to new execution data to tackle the scenario where train and test data have different distributions.
- We show how an index tuner can use this classifier to significantly improve its recommendations while still being "in-sync" with the optimizer.
- We present an extensive evaluation of our techniques using a diverse collection of benchmarks and real workloads.

2 OVERVIEW

2.1 Index Recommendation

The problem of index recommendation is to find the best configuration for a workload, often under a storage budget:

Problem Statement 1. ***Index tuning**: Given a workload $\mathcal{W} = \{(Q_i, s_i)\}$, where Q_i is a query and s_i is its associated weight, and a storage budget B, find the set of indexes or the configuration C that fits in B and results in the lowest execution cost $\sum_i s_i \cdot cost(Q_i, C)$ for $\mathcal{W}$, where $cost(Q_i, C)$ is the cost of query Q_i under configuration C.*

Since the cost of query and the size of index are unknown during index tuning, state-of-the-art index tuners use optimizer's estimates.

Index tuners can support additional constraints, such as limit the number of recommended indexes. In particular, to avoid query regression, we constrain $cost(Q, C) \leq (1 + \lambda) \cdot cost(Q, C^0)$ for all queries, where C^0 is the database's initial configuration and λ is the regression threshold (e.g., 0.2).

In production systems, physical configurations are changed incrementally and continuously to constrain the impact on workload, limit the chances of major regressions, and adapt to changing workloads. In this case, the index tuning is invoked iteratively to find the best indexes:

Problem Statement 2. ***Continuous index tuning**: Given the number of iterations K, a workload $\mathcal{W} = \{(Q_i, s_i)\}$, where Q_i is a query and s_i is its associated weight, and a storage budget B, find a sequence of configurations $C^1 \cdots C^K$, where the change in configuration $C^k - C^{k-1}$ fits in B at each iteration k and $\sum_{k=1}^{K} \sum_i s_i \cdot cost(Q_i, C^k)$ results in the lowest execution cost for $\mathcal{W}$.*

Since finding the optimal sequence of configurations requires exhaustive search which is prohibitively expensive [3, 12], state-of-the-art index tuners greedily find the best C^k at each iteration k. Similarly, we can limit the number of indexes recommended per iteration and constrain that no query regresses, i.e., $cost(Q, C^k) \leq (1 + \lambda) \cdot cost(Q, C^{k-1})$ for all queries Q and iterations k.

2.2 Classification Task

Given two query plans $\mathcal{P}_1$ and $\mathcal{P}_2$ for query Q chosen by the query optimizer under configurations C_1 and C_2, our goal is to predict if $\mathcal{P}_2$ is cheaper or more expensive in execution cost compared to $\mathcal{P}_1$. Given configurable thresholds $\alpha_1 > 0, \alpha_2 > 0$, $\mathcal{P}_2$ is more expensive if $ExecCost(\mathcal{P}_2) > (1 + \alpha_1) \cdot ExecCost(\mathcal{P}_1)$ and cheaper if $ExecCost(\mathcal{P}_2) < (1 - \alpha_2) \cdot ExecCost(\mathcal{P}_1)$, where $ExecCost$ is the execution cost of a plan. For simplicity, we use $\alpha_1 = \alpha_2 = \alpha$ which is usually set to 0.2 to specify the significance of the change.

We set up a **ternary** classification task as: given $\langle \mathcal{P}_1, \mathcal{P}_2 \rangle$, assign label **regression** if $\mathcal{P}_2$ is more expensive, **improvement** if $\mathcal{P}_2$ is cheaper, and **unsure** otherwise. A binary classifier that flags regression or non-regression is sufficient to enforce no regression constraint, a ternary classifier supports additional properties for single query tuning (see Section 5).

We assign labels by comparing the logical execution cost of plans, e.g., the CPU time or number of bytes processed. Such logical execution cost is more robust to runtime effects, such as concurrency, compared with a physical measure (e.g., latency). Due to measurement variance or different parameters for a query template, we assign labels with a robust statistical measure, e.g., the median of several executions.

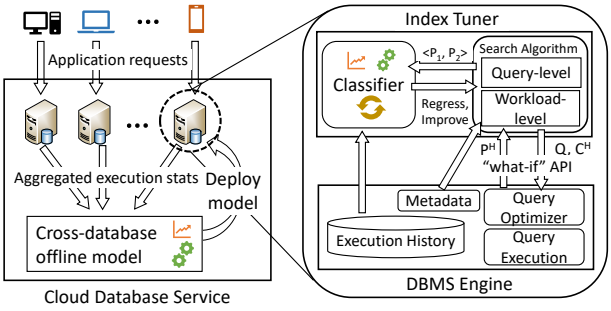


Figure 2: Overview of an architecture leveraging the classifier trained on aggregated execution data from multiple databases in a cloud database service.

2.3 Architecture

Figure 2 illustrates the end-to-end architecture of an index tuner leveraging aggregated execution data in a cloud platform, such as Azure SQL Database. Databases emit aggregated query plan execution statistics through telemetry. These statistics include the featurized plans (see Section 3.2) from different index configurations, which happen naturally as indexes are changed by human administrators or automated services [24, 52]. Plans of the same query are identified with a unique query hash generated by Azure SQL Database from the abstract syntax tree. Using different plans for the same query, we construct the pairs that provide the final feature vectors (see Section 3.3) and use plans' execution cost to assign the class label. We train a classifier by aggregating this data across databases, which corresponds to the *offline* model (Section 4.1). This offline model is then deployed for the index tuner to use during its search for index configurations (Section 5). The application's queries execute in parallel to the index tuner implementing its indexes, similar to the auto-indexing service in Azure SQL Database [24]. These new configurations result in additional execution data collected local to each database. We *adapt* the model to each database (Section 4.3) by retraining the adaptive model on these additional data at each invocation of the index tuner.

3 FEATURIZING QUERY PLAN PAIRS

3.1 Rationale

The input to the classification task is a pair of query plans, where each query plan is a tree of operators. Off-the-shelf ML techniques rely on feature vectors of fixed dimensions. Hence, we need to convert the pair of query plans into feature vectors. Several approaches exist to convert graphical or tree-structured data into vectors [33, 71]. While being more general purpose, such approaches do not leverage information specific to database semantics and critical for execution cost that query plans encode, e.g., operator type, parallelism, execution mode, etc. To better leverage such information, we develop a featurization, inspired by other approaches used for DB applications (e.g.,[5, 31]), that is efficient and results in models with high accuracy. While some featurization aspects leverage the details of SQL Server, we expect the key ideas are applicable to other database engines.

We use the following guiding principles to encode the semantics of the query plans and other factors that contribute to execution cost in our featurization:

- **Learn across queries and databases:** Be schema agnostic to allow cross-database learning, leveraging execution data from millions of databases in a cloud platform.
- **Learn from the optimizer:** Leverage the valuable information presented in query plans generated by industrial-strength query optimizers.
- **Learn from information in estimated query plans:** The index tuner can search for configurations that have never been implemented. Thus, execution statistics, such as true cardinalities, are rarely available at inference time. Featurization should not use such information.

In addition to the above principles, the feature vectors should encode the following key types of information that provide a way for the model to learn the classification task:

- **Measure of work done:** The query optimizer's estimate for an operator's cost or the number of rows processed by an operator are example features for this measure.
- **Structural information:** Join orders or the position of an operator in the plan is often useful, especially when comparing two plans for the same query.
- **Physical operator details:** Physical operators in a plan play a crucial role in the cost. For instance, a nested loop join will have very different cost compared to a merge join even if they correspond to the same logical join operator.

We assume all execution data is collected on similar hardware; extensions to heterogeneous hardware is future work.

3.2 Featurizing a Plan

Guided by our principle of being able to learn across plans, queries, and databases and our goal of capturing physical operator details, we use the physical operators supported by SQL Server as our feature dimensions or attributes. SQL Server supports a set of physical operators (such as Index Scan, Table Scan, Hash Join, etc.) which are known in advance and also do not change very frequently. A query plan is a tree of these physical operators. Two additional properties of physical operators are relevant to execution cost: (*a*) *parallelism*: whether the operator is single-threaded (serial) or multi-threaded (parallel); and (*b*) *execution mode*: whether the operator processes one row at a time (*row mode*) or a batch of data items in a vectorized manner (*batch mode*). Each operator is assigned a **key** of the form

Table 1: Example feature channels with different ways of weighting nodes encoding different types of information. All estimates are from the query optimizer.

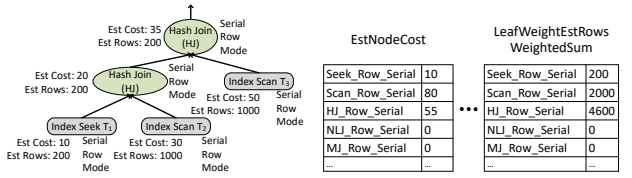
Channel	Description
EstNodeCost	Estimated node cost as node weight (work done).
EstRowsProcessed	Estimated rows processed by a node as its weight (work done).
EstBytesProcessed	Estimated bytes processed by a node as its weight (work done).
EstRows	Estimated rows output by a node as its weight (work done).
EstBytes	Estimated bytes output by a node as its weight (work done).
LeafWeightEstRowsWeightedSum	Estimated rows as leaf weight and weight sum as node weight (structural information).
LeafWeightEstBytesWeightedSum	Estimated bytes as leaf weight and weight sum as node weight (structural information).

⟨Physical Operator⟩_⟨Execution Mode⟩_⟨Parallelism⟩. Execution mode is either *Row* or *Batch*, and parallelism is either *Serial* or *Parallel*. Examples of such keys are ⟨Seek_Row_Serial⟩, ⟨HashJoin_Row_Serial⟩, ⟨Scan_Batch_Parallel⟩, etc. Since the set of physical operators is fixed, the set of keys is also fixed.

For a given query plan, we assign a value to each key which: (*i*) measures the amount of work done by the corresponding operators in the plan; (*ii*) encodes structural information. In a plan with multiple operators having the same key, we sum up all the values assigned to the key. If an operator does not appear in a plan, we assign zero to the corresponding key, allowing a fixed dimensionality of the vector. Different ways of assigning a value to an operator encode different information and create different **feature channels**. Table 1 lists the different feature channels, how the weights are computed, and what information they encode. Each channel has the same dimensionality. Since channels in Table 1 have some redundancy, a subset of channels, usually two or three, are sufficient as long as we pick channels that encode a measure of work and structural information. We also use the optimizer-estimated plan cost as a feature.

Table 1 shows various ways to encode the amount of work done by an operator, such as using the optimizer's estimate of the node's cost (*EstNodeCost*) or the estimated bytes processed by the node (*EstBytesRead*). The channels with *WeightedSum* suffix encode some structural information even in the *flattened* vector representation. We assign a weight to each node which is computed recursively from the leaf nodes in the plan to the root. Each leaf node has a weight, e.g., estimated number of rows output by the node. Each node has a height, i.e,. starting with 1 for the leaves and incremented by 1 for each level above the leaf. The value of a node is the sum of weight × height of all its children. Structural changes in the plan, e.g., join order change, will likely result in different children weights and potentially node heights (e.g., see Figure 4), thus resulting in different feature vectors.

Figure 3 gives an example of our featurization for a simple query plan shown in Figure 3(a). The plan joins three tables



(a) Example query plan. (b) Feature channels for the plan.

Figure 3: An example of encoding a query plan into a vectorized representation called feature channels.

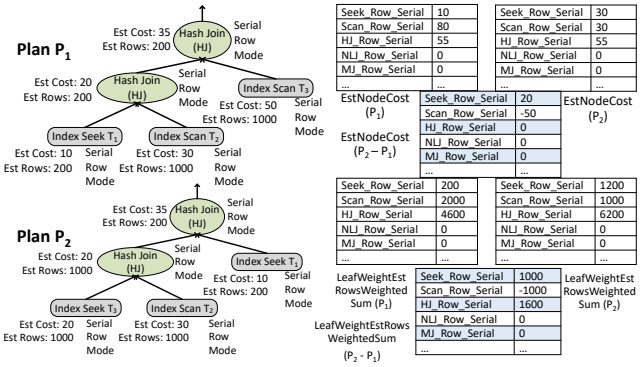


Figure 4: Example of combining the individual plan features into a feature vector for the pair by using a channel-wise difference. Join order change (a structural change) is reflected in the values for channels ending with *WeightedSum*.

and returns their result, executing single-threaded in row mode. Each node is also annotated by the physical operator being used as well as some optimizer-estimated measures, such as estimated node cost, estimated rows, etc. Figure 3(b) shows how two example channels are computed using the raw values obtained from the plan. Consider the *EstNodeCost* channel, it uses the optimizer-estimated node cost as the weight and sums the weights of the same key in the plan. For instance, the keys ⟨Scan_Row_Serial⟩ corresponds to two operators and the weight in Figure 3(b) for this key is the sum of the weight, 50 and 30, of each operator.

3.3 Featurizing a Pair

After featurizing individual plans, we combine their features to encode a pair of plans $\langle \mathcal{P}_1, \mathcal{P}_2 \rangle$. A key insight driving this combination is that the classifier is conceptually learning to find the difference between the plans. If we rewrite the expression in Section 2, a regression label is assigned is if:

$$\frac{ExecCost(\mathcal{P}_2) - ExecCost(\mathcal{P}_1)}{ExecCost(\mathcal{P}_1)} > \alpha \qquad (1)$$

Empirically, we observe that if the featurization mimics this mathematical transformation used to assign labels, it improves the model's prediction performance compared to just concatenating the selected channels from both plans.

The first mathematical transformation we consider is computing an attribute-wise difference among the corresponding channels from $\mathcal{P}_1$ and $\mathcal{P}_2$; we refer to this transformation as **pair_diff**. Figure 4 illustrates this process for a pair of plans using the *EstNodeCost* and *LeafWeightEstRowsWeightedSum* channels. To further mimic the formula (1), a different transformation is to compute the attribute-wise difference for each channel and then divide it by the corresponding attribute in $\mathcal{P}_1$. We call it **pair_diff_ratio**. To overcome the challenge where many attribute values in a channel might be zero, we clip values to a large enough value (e.g., 10^4) on division by zero; such clipping is commonly-used in ML [32]. An alternative to avoid the need of clipping is to use a different denominator. For every channel, we compute the sum of the values of all attributes and use it as the denominator and the attribute-wise difference as the numerator. We refer to this transformation as **pair_diff_normalized**.

4 LEARNING THE CLASSIFIER

4.1 Offline Model

Once the query plan pairs have been featurized as described in Section 3, we can use off-the-shelf ML packages to train the classifier. The simplest approach is to use linear or tree models to train one *offline* model. This offline model can be per-database or even a global model for all databases. Among linear learners, we picked Logistic Regression (**LR**) for its simplicity and training speed. Among the tree-based techniques, we tried two broad categories of models, both of which are ensemble of trees: (*a*) *bagging* ensembles such as Random Forest (**RF**) [9]; (*b*) *boosting* ensembles such as Gradient-boosted Trees (**GBT**) [21] and Light GBM with gradient-boosted decision trees (**LGBM**) [42].

4.2 Need for Adaptation

A crucial assumption for ML models to perform well is that the train and test data follow the same distribution in the feature space. This assumption often does not hold in our setting, because train/test data distributions differ across databases and even within a database across queries. This is empirically confirmed by our analysis on a large collection of benchmarks and customer workloads (see Section 7.7).

There are three major reasons for these differences. First, in a cloud platform where a huge variety of new applications get deployed every day, the execution data on a new database can be completely different from that observed for existing databases. Second, within a database, there is huge diversity in the types of queries executed (e.g., the joins, predicates, aggregates), and changing indexes adds even more diversity. Third, even with databases where several plans of a query have executed, these plans often represent a tiny fraction of the alternatives considered during the tuner's search. For complex queries, the tuner may explore hundreds of different configurations that may result in tens of very different plans. The featurized representations of these unseen plans can be significantly different from the executed ones.

Preparing for unseen data is a necessity in our setting. While the offline models can be periodically retrained with new data and redeployed in a cloud platform, given the scale of the infrastructure, such as Azure SQL Database, the retraining and redeployment will be infrequent (e.g., once a day or even less frequent). Thus, a model that quickly adapts to new execution data in a database is invaluable for automated indexing which continuously tunes and changes indexes.

4.3 Adaptive Model

A simple approach to *adapt* a model to a database is to learn a *local* model with execution data only from that database and use it for prediction. This local model is lightweight, usually trained on hundreds or a few thousands of plans, and is orders of magnitude cheaper than retraining the offline model with the new data. However, the local model is only trained on a small amount of execution data from a database, and it can overfit and generalize poorly to unseen queries and plans even from the same database. Completely ignoring the offline model trained from millions of plans on other databases can be problematic especially when the execution data from this database is insufficient. Thus, we explore approaches to combine the offline and local models.

Nearest neighbor: We observe that ML models have higher prediction accuracy for points which are in the neighborhood (in feature space) of training data. This is because data points with similar feature values likely have similar labels. Thus, for each data point d, if the local model has trained with data in d's neighborhood, we can use the local model to predict; otherwise, we use the offline model. This approach requires us to tune the distance threshold parameter for neighborhood and adds the overhead of finding the neighbors.

Uncertainty: ML models often expose an uncertainty score where lower uncertainty corresponds to a higher probability of correct prediction [9]. Thus, another approach is to use uncertainty scores from the local and offline models, and picks the prediction with lower uncertainty. While intuitive, the semantics and robustness of these uncertainty scores are model-specific and can be a weak indicator.

Meta model: Nearest neighbors and uncertainty provide weak signals about which of the online and offline models is more likely to predict correctly for a given data point. Finding appropriate thresholds and combinations of these

signals to make the choice is itself a learning problem. Hence, instead of manually tuning these thresholds, we formulate a *meta learning* problem where we train a *meta model* to use predictions from both the models, the uncertainty, and neighborhood as features to make the final prediction.

We use the local execution data collected on a database to train the meta model M_{meta}. We split the data into two disjoin sets D_l and D_m. We use D_l to train the local model (M_{local}) and use D_m to train the meta model (M_{meta}); the offline model $M_{offline}$ is trained with execution data from other databases. We extract *meta features* for each data point d in D_m, such as the predictions of $M_{offline}$ and M_{local} for d, the corresponding uncertainty scores, and the distances and labels of close neighbors of d in D_l. With these meta features, we train a Random Forest on D_m. Both M_{local} and M_{meta} are retrained as new execution data becomes available.

5 INTEGRATION WITH INDEX TUNER

Section 2.3 describes the overall end-to-end architecture of an index tuner leveraging the aggregated execution statistics. We now describe in detail how to integrate a classifier into the index tuner. The model, trained with a pre-specified threshold α, supports two APIs: (i) `IsImprovement` ($\mathcal{P}_1, \mathcal{P}_2$); and ($ii$) `IsRegression` ($\mathcal{P}_1, \mathcal{P}_2$); where $\mathcal{P}_1$ and $\mathcal{P}_2$ are plans for query Q. This interface is independent of the model type or whether it is the offline or adaptive model.

We implement an index tuner similar to state-of-the-art tuners [2, 17, 65]. Given a database $\mathcal{D}$, a workload $\mathcal{W}$, an initial configuration C^0 when the tuner is invoked, and constraints such as the maximal number of indexes recommended, a storage budget, or no query regresses compared to its cost in C^0, the tuner recommends a new configuration C^R to minimize the total query optimizer-estimated cost of $\mathcal{W}$ subject to these constraints. The tuner has two major phases: (a) a **query-level** search to find the optimal configuration for each query $Q \in \mathcal{W}$; and (b) a **workload-level** search to find the optimal configuration for $\mathcal{W}$ by enumerating different sets of indexes obtained from phase (a).

We retain the tuner's use of the "what-if" API to obtain the plan $\mathcal{P}^H$ for a hypothetical configuration C^H to be "in-sync" with the optimizer [18]. For query-level search, we use `IsRegression` to enforce the no regression constraint. The tuner only considers a hypothetical configuration C^H as a candidate configuration for query Q_i if `IsRegression`($\mathcal{P}_i^0$, $\mathcal{P}_i^H$) is **false**, where $\mathcal{P}_i^H$ is the plan in C^H and $\mathcal{P}_i^0$ is the plan for the initial configuration C^0. Optionally, the tuner can only consider configurations that are predicted to improve execution cost. That is, it will update the cheapest configuration C it has searched so far and the corresponding plan $\mathcal{P}_i$ to C^H and $\mathcal{P}_i^H$ if `IsImprovement`($\mathcal{P}_i^H$, $\mathcal{P}_i$) is **true**. Since the classifier is trained with $\alpha > 0$ (e.g., 0.2), for certain plan pairs, both `IsImprovement` and `IsRegression` could return false, which corresponds to the *unsure* class of insignificant difference in execution cost. In such cases, the tuner uses a cost model, e.g., the optimizer.

Similarly, for workload-level search, we use `IsRegression` API to enforce the no regression constraint for each $Q \in \mathcal{W}$. That is, the tuner searches for configuration C^H that minimizes the optimizer's estimated cost of $\mathcal{W}$ and also satisfies that `IsRegression` ($\mathcal{P}_i^0$, $\mathcal{P}_i^H$) is false for all $Q_i \in \mathcal{W}$.

6 DESIGN ALTERNATIVES

6.1 Alternative Learning Tasks

While we focus on training a classifier to predict query regressions, an alternative is to use an ML regression model. We consider three types of regressors: (a) an **operator-level regressor** that given a plan, estimates the execution cost of each operator and then determines the plan's execution cost; (b) a **plan-level regressor** that learns to predict the execution cost of a plan; (c) a **plan pair regressor** that given a pair of plans $\langle \mathcal{P}_1, \mathcal{P}_2 \rangle$, learns to predict the ratio $\frac{\text{ExecCost}(\mathcal{P}_2)}{\text{ExecCost}(\mathcal{P}_1)}$. The regressors ($a$) and ($b$) estimate execution cost and can be used for other applications including comparing plans and workload costs. The regressor (c) is more specific to comparing plans, but can support any threshold α, while our classifier needs to be trained for a pre-specified α.

We use the operator-level model proposed by Li et al. [49] for the regressor in (a). We ensure that our implementation has comparable performance to that reported in the paper. The plan-level regressor (b) is similar to Akdere et al. [5] adapted to our setting. We use the feature channels described in Section 3.2 as they are more comprehensive for the complex execution model and class of operators in SQL Server.

The plan-pair regressor (c) uses the featurization in Section 3.3. Since there can be several orders of magnitude difference in values of the ratio, to make the learning problem easier and minimizing the L_1 error meaningful, we label the pairs as $\log(\frac{\text{ExecCost}(\mathcal{P}_2)}{\text{ExecCost}(\mathcal{P}_1)})$. In addition, since for comparing plans, we usually care for α in the range of 0.2 to 0.3, we clip the ratio to be in the range of 10^{-2} to 10^2.

6.2 Alternative Classifiers

6.2.1 Deep Neural Networks. Over the past decade, Deep Neural Networks (**DNNs**) have achieved success with complex cognition tasks, e.g., object detection, tracking, etc [32]. Hence, it is natural to explore if DNNs can provide a significant improvement in prediction performance compared with ML techniques in Section 4. Moreover, theoretically DNNs can learn non-linear functions on the input features [36, 37], which could potentially complement our featurization.

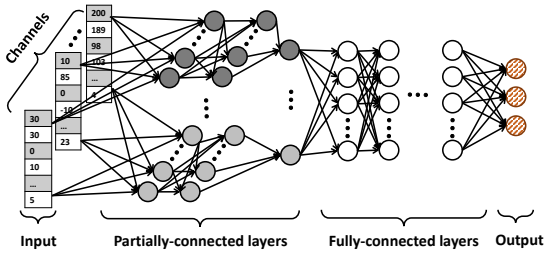


Figure 5: A partially-connected DNN architecture.

Since DNNs cannot automatically featurize query plans, we use the same featurization of plans in Section 3 as the input. Since there are no well-known network architectures for our problem, we explored a variety of network architectures:

Fully-connected networks. We start with a fully-connected network architecture: all neurons in a layer receive inputs from all neurons in the previous layer. Each neuron uses a non-linear activation [32], and the output layer uses a softmax activation to learn class labels using one-hot encoding. We configure the network to maximize categorical cross entropy. Because our featurization can have hundreds of dimensions, the number of trainable parameters grows rapidly to hundreds of thousands with more layers and the training becomes prohibitively expensive. We further observed that the prediction performance of such a network was low even on the training data, in spite of tens of hidden layers.

Partially-connected networks. To reduce the number of trainable parameters, we design a novel *partially-connected* network architecture. Our key intuition is that when learning to compare plans, it is more meaningful to compare similar operator types (or the same keys in our featurization in Section 3.2) across different channels. For example, when comparing two plans with joins, it is natural to compare the values of hash joins in both plans instead of hash join in one plan and index scan in another. As shown in Figure 5, by removing cross-key connections in early layers, a partially-connected network can first learn to combine values within a key. The last layer in the partially-connected part reduces to one neuron per key, followed by fully-connected layers to learn to combine different keys for the final prediction.

Training deeper networks. While deeper networks can learn more complex functions in theory [32, 36, 37], the difficulty in training them properly can lead to worse performance compared to shallower networks [32, 35, 59]. This challenge comes from: (*i*) propagating gradients backwards across layers of non-linear activations; and (*ii*) vanishing gradients (i.e., gradients become infinitesimally small).

We faced the same challenge and found two techniques effective for combating the issue: skip connections [35] and highway networks [59]. We incorporated these techniques into the fully-connected layers in our partially-connected network architectures.

6.2.2 Hybrid of DNN and Tree. The hidden layers of DNNs can be conceptually viewed as learning a function to combine the input feature vector into a latent representation, i.e., the output of the last hidden layer. The output layer then learns a model using the latent representation. For the classification task, the output layer is often equivalent to Logistic Regression (when using either the *sigmoid* or *softmax* activation for the output layer). We explore the design of a *Hybrid DNN* where the output layer of a DNN is replaced by a more advanced model, e.g., a Random Forest (RF). We stack an RF over a DNN to use the featurization power of the DNN with RF's ability to efficiently learn complex rules. We first train the DNN as before and then train the RF by taking the output from the last hidden layer of the DNN as the new input. At inference time, we first use the DNN to infer with the input and then feed the output of its last hidden layer into the RF for the final prediction.

6.2.3 Adapting DNNs. To adapt the DNN models to new data distribution, we use *transfer learning* [53] to transfer the knowledge from the offline model and customize the predictions to a specific database. We initialize a DNN with pre-trained weights from the offline model. We then retrain the weights of the output layer (or the last a few layers) with back-propagation on the new training data, while freezing the weights of the other hidden layers. This training is fast since the amount of additional training data is small. For individual DNNs explored in Section 6.2.1, we only retrain the output layer. For Hybrid DNN in Section 6.2.2, since the stacked RF is essentially equivalent to the output layer, we retrain the RF while freezing the weights of the DNN.

7 EXPERIMENTAL EVALUATION

The major facets our evaluation focuses on are:

- **Regression vs. classification**: evaluate whether a classifier is more accurate than a regressor for the task of comparing plans. *We observe between* $2\times - 5\times$ *reduction in fraction of errors for picking the cheaper one between a pair of plans when using the classifier compared to using other learned or analytical models to predict cost or cost ratio.*
- **Offline model**: evaluate the performance and trade-offs of different ML techniques for offline learning across databases, queries, and plans. *We find that overall the RF-based models outperform others in accuracy and training efficiency.*
- **Need for adaptation and adaptive model**: empirically quantify the need and the benefits for adapting to unseen databases and queries. *We observe that the lightweight adaptive strategies are effective in reducing the fraction of*

Table 2: Aggregate statistics about the schema and query complexity of the workloads.

Workload	DB size (GB)	# tables	# queries	Avg. # joins	Total # Plans	Max # Plans / Query	# Plan Pairs
TPC-DS 10g	11.2	24	92	7.9	3,500	166	200,825
TPC-DS 100g	87.7	24	92	7.9	3,714	139	211,541
TPC-H 10g Zipf	12.1	8	22	2.8	299	44	6,986
TPC-H 100g Zipf	132	8	22	2.8	306	49	6,600
Customer1	87.7	20	111	5.9	4,669	114	144,474
Customer2	1723	23	34	7.2	2,364	174	214,842
Customer3	44.6	614	32	8.1	584	50	17,926
Customer4	1.2	8	125	1.6	2,539	139	153,752
Customer5	283	3,394	35	7.2	2,041	170	133,671
Customer6	9.9	474	311	21	18,677	140	865,125
Customer7	93	22	23	5.2	841	82	37,157
Customer8	0.25	129	474	1.1	3,746	135	82,738
Customer9	48.7	7	15	2.2	76	18	680
Customer10	17.0	32	10	8	242	46	7,854
Customer11	7.2	81	399	0.8	2820	74	19,257

errors in plan comparison by up to 2× with a tiny fraction of newly-labeled plans from the unseen database.

- **Improvement in index recommendations**: quantify the end-to-end recommendation improvement in execution cost by using the classifier. *We observe that augmenting the index tuner with the model eliminates most regressions while preserving or even boosting the improvement.*

In Appendix A, we present additional experiments on production data, feature sensitivity, different DNN architectures, and index recommendation quality with different models.

7.1 Metrics

We measure the classifier's prediction quality using Precision, Recall, and F1 score [9]. These metrics are robust to skew in distribution of classes. For the regression class, plan pairs labeled with that class are considered positives while others are negatives. For a given test set of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN), Precision is $(\frac{TP}{TP+FP})$, i.e., the model's accuracy of positive prediction; Recall is $(\frac{TP}{TP+FN})$, i.e., the model's coverage in correctly predicting the positives; F1 score is the harmonic mean of precision (P) and recall (R) $(\frac{2PR}{P+R})$.

7.2 Workloads

We use a diverse collection of workloads including industry-standard benchmarks, such as TPC-H [64] and TPC-DS [63], and eleven workloads from customers of SQL Server. For TPC benchmarks, we use two different scale factors, 10 and 100, which share query templates but use different parameters, data sizes, and distributions. For TPC-H, we use a skewed data generator [54] instead of uniform, which makes cost estimation more challenging. Table 2 summarizes the key statistics of the workloads which comprises of SELECT only.

7.3 Experimental Setup

To thoroughly evaluate the models in the huge search space a tuner explores, we study the models in isolation by simulating plan pair cost comparisons outside the tuner. Our training and test data is derived from workloads in Table 2. Since our goal is to improve index recommendations, we collect execution data for a diverse set of index configurations. We use index recommendations generated by the tuner (without the ML model) for individual queries in the workload. To simulate the configurations and plans during the tuner's search, we construct different subsets of indexes from the tuner's recommendation. We implement those indexes, execute the queries in isolation with data entirely in-memory, and record query plans and execution cost, e.g., CPU time. We use the median CPU time over several executions for labeling. To simulate different initial database configurations (the input C^0 to the tuner), we use different initial indexes, e.g., without any indexes, with B+ tree indexes, and with columnstore indexes. Table 2 summarizes the statistics of the execution data, such as the number of plans, the number of plan pairs, etc. For complex queries, there can be hundreds of plans due to hundreds of index subsets from a large set of indexes recommended by the tuner. We also evaluate the models with production data in Appendix A.1, where the execution data is collected passively as described in Section 2.3.

Since the task is to compare plan costs, we construct our data set as pairs of plans $\langle \mathcal{P}_1, \mathcal{P}_2 \rangle$ for the same query Q. We split the data in different ways to construct the train/test sets. (i) **Pair**: split the union of all plan pairs into disjoint sets; (ii) **Plan**: split the set of plans into two disjoint sets of plans from which the pairs are constructed. Plans in test are different from plans in training, simulating the setup where during the tuner's search, the model is used to infer on new plans for new configurations. (iii) **Query**: split the set of queries into two disjoint sets, and use the plans corresponding to queries in each set to construct the pairs. This corresponds to inference on new queries that did not appear in training. (iv) **Database**: the test set is an unseen database for which no queries or plans appear in training. With these approaches of splitting, the train/test distributions become increasingly different, with most similarity for Pair and least for Database.

For splitting by pair, plan, and database, we randomly split train/test and repeat the experiment five times; since there is more diversity among queries, for splitting by query, we repeat the experiment ten times. We report the average F1 score for the regression class; the F1 score for the improvement class is comparable. The Precision and Recall numbers are close to the F1 score in most cases and are omitted for brevity. Unless otherwise stated, we use the *EstNodeCost, LeafWeightEstBytesWeightedSum* feature channels and pair_diff_normalized to combine features of a plan pair.

7.4 Hyper-parameter Tuning

We use standard cross-validation to tune hyper-parameters. Following are the hyper-parameters for tree-based techniques: (*i*) **Number of trees in the ensemble.** We vary the number of trees from 50 to 400 and observe very little change in F1 score. (*ii*) **Regularization.** (*a*) minimum number of samples in a leaf node (i.e., 1), and (*b*) early stopping threshold that will prevent node splitting if the Gini impurity [9] is below a threshold (i.e., 10^{-6}).

We report the most significant hyper-parameters for DNNs: (*i*) **Activation function.** We find *tanh* activation results in much better training and prediction accuracy compared to *ReLU* [32, 34, 35], *Leaky ReLU* [32], or *Parameterized ReLU* [34]. (*ii*) **Initialization.** We find a clipped normal distribution [34] as the most effective initialization strategy for weights, with biases initialized to zero. (*iii*) **Regularization.** We use two regularization techniques: (*a*) dropout regularization (with dropout ratio of 0.2); and (*b*) L_2 regularization of the weights (with a regularization factor of 10^{-3}). (*iv*) **Learning rate.** We find using an adaptive learning rate to be more effective than using a fixed learning rate. We start with a learning rate of 0.01 and then once the loss does not improve over several epochs, e.g., hit a plateau or a saddle point, we halve the learning rate. We allow up to 10 such adaptations. (*v*) **Optimizer.** We use the *Adam* optimizer [45].

7.5 Regression vs. Classification

We now empirically evaluate our key insight that a classifier is more accurate in predicting regressions or improvement compared to models predicting execution cost or cost ratios. We compare the regressors discussed in Section 6.1 with the classifier using the query optimizer's cost estimates. For the plan-level model (*Plan Model*), we tune hyper-parameters and choose the model configuration with the least L_1 loss ($|EstCost - ActualCost|$) in cross-validation, i.e., an RF-based model with 250 trees and feature channels *EstNodeCostSum, EstDataSizeReadSum, LeafWeightEstBytesWeightedSum*. The plan pair model (*Pair Model*) uses a GBT-based model with 250 trees and the same feature channels as *Plan Model*, and the plans are combined using pair_diff_ratio. For the classifier (*Classifier*), we use an RF-based classifier as a representative among classifiers we have studied. We use 250 trees and feature channels *EstNodeCostSum, LeafWeightEstBytesWeightedSum* where the plans are combined using pair_diff_normalized. All the models are trained using ML.NET [51]. The operator-level model (*Operator Model*) is implemented as described in Section 6.1.

Figure 6 reports the F1 score for different approaches. In each cluster of bars, *Optimizer* uses the optimizer's estimates to compare, the second to fourth bars correspond to the different regressors, and the last bar corresponds to the classifier.

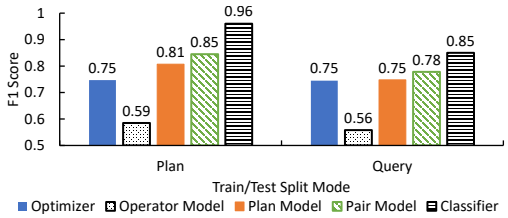


Figure 6: F1 score of different approaches to compare execution costs of a pair of plans.

Table 3: Segmented F1 score for different models, i.e., Optimizer (O), Pair Model (P), and Classifier (C), with the best F1 score for each segment in bold.

Diff Ratio	0.2 – 0.5			0.5 – 1			1 – 2			> 2		
Plan Cost	O	P	C	O	P	C	O	P	C	O	P	C
0-25%				0.70	**0.84**	**0.84**	0.74	0.92	**0.93**	0.85	0.96	**0.97**
25-50%	0.53	0.71	**0.75**	0.63	0.87	**0.89**	0.73	0.92	**0.94**	0.92	0.97	**0.99**
50-75%	0.53	0.77	**0.84**	0.62	0.90	**0.93**	0.71	0.95	**0.97**	0.92	0.98	**0.99**
75-100%	0.50	0.70	**0.81**	0.57	0.86	**0.89**	0.67	0.93	**0.94**	0.92	0.96	**0.99**

Different clusters correspond to how the train and test data are split, i.e., split by Plan or Query. The train set has 60% of the plans or queries, while the test set has the remaining 40%. As is evident, the classifier's F1 score is significantly higher compared to any other model. In particular, *compared with the query optimizer, which is used in state-of-the-art index tuners, for unseen plans, the classifier remarkably increases the F1 score by* 21 *percentage points, equivalent to about* 5× *reduction in the error. For unseen queries, which is a much harder problem to predict, the classifier still improves over the optimizer by* 10 *percentage points, i.e., almost* 2× *reduction in error. Moreover, the classifier is much more accurate compared to any of the regressors.* Interestingly, the operator-level model has a significantly lower F1 score compared to the plan-level model, even though it has significantly lower $L1$ error in CPU cost estimation. This is further evidence for our observation that the ML models need to be trained to minimize the loss function that directly impacts the application, e.g., plan cost comparison in our setting. Even if the $L1$ error for individual plans may be low, the errors might be in opposite directions or large enough to result in an incorrect comparison outcome. Similarly, even if the $L1$ error for cost ratio may be low, the predicted ratio can still be opposite in direction and result in incorrect comparison prediction.

The averaged F1 score does not differentiate the errors by how expensive a plan is or how significant the regression is, while an error in more expensive plans or more significant regressions are worse than cheaper plans and smaller regressions for index tuning. Thus, we break down the result by splitting the data into segments by the percentile of the cost of a plan pair (Plan Cost, i.e., $cost_1 + cost_2$) and

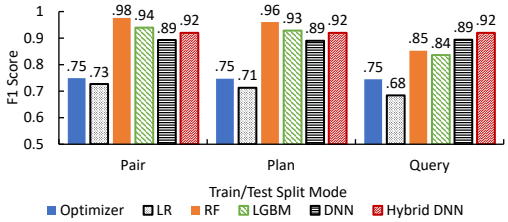


Figure 7: Comparison of different modeling techniques for the classification task.

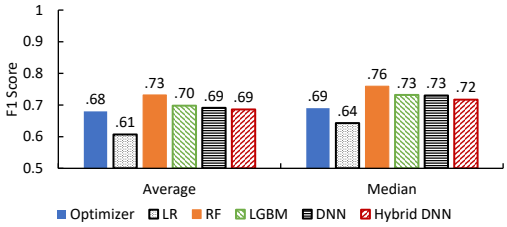


Figure 8: F1 score on unseen databases.

the ratio of plan cost difference of a plan pair (Diff Ratio, i.e., $max(cost_1, cost_2)/min(cost_1, cost_1) - 1$). We compare the performance of all the models and report three of them due to space limit: *Optimizer (O)*, *Pair Model (P)*, and *Classifier (C)*. We choose the *Pair Model* since it has the best F1 score among the three regressors presented earlier. Table 3 shows the segmented F1 score, with the best F1 score in bold for each segment. As is evident, the classifier outperforms the other models in all segments, especially when the cost difference ratio is small to moderate, i.e., < 1. In Appendix A.2, we simulate the workload cost with models' predictions. Again, using the classifier results in the lowest workload cost.

7.6 Offline Model

We compare the performance of all the classes of models we consider in the paper (see Section 4.1 and Section 6.2): Logistic Regression (LR), Random Forest (RF), Gradient-boosted methods (LGBM), Deep Neural Networks (DNN), and the hybrid of DNN and RF (Hybrid DNN). Here we report the test F1 score of the models with best cross-validation F1 score. The tree and linear models are trained using the scikit-learn library [57] while the DNN models are trained using Keras [43] on Tensorflow [1]. The scikit-learn models are similar in performance to the ML.NET models used earlier.

Figure 7 reports the performance of the different classifiers with different modes of splitting the train and test data: Pair, Plan, and Query. The tree-based models use 400 trees, and the RF model uses a Gini improvement threshold of 10^{-6}. The DNN is a partially-connected network with 3 hidden layers for the partially-connected part, 12 hidden layers for the fully-connected part, and 64 neurons per hidden layer. This network gives the best cross-validation performance among various partially-connected DNNs we have studied, and a larger network does not further improve the performance. The Hybrid DNN stacks an RF with 50 trees over the partially-connected DNN described earlier. All techniques use *EstNodeCostSum, LeafWeightEstBytesWeightedSum* feature channels and combine plans with pair_diff_normalized.

For unseen pairs and plans, where train and test distributions are more similar, the tree models have better F1 score, with RF being the best. The DNNs, both individual and hybrid, outperform the tree models for unseen queries, with Hybrid DNN being the best. The DNNs also take longer to train. For our data set with about two million plan pairs, the RF trains in tens of minutes and infers in less than 10 microsecond per data point in average on a commodity Intel Xeon processor with 20 CPU cores. On the other hand, the DNN trains in a couple of hours and infers in tens of microseconds on an NVIDIA Tesla *P*100 GPU. Both models consume tens of megabytes of memory.

7.7 Need for Adaptation

We now consider a more drastic data shift by holding out databases. From the fifteen databases in our setup (see Table 2), we train on fourteen and test on the remaining one. We repeat this experiment for all fifteen databases and report the aggregate F1 score across all fifteen runs.

Figure 8 plots the F1 score across all the databases for all the models, i.e., LR, RF, LGBM, DNN, and Hybrid DNN, with the optimizer as a baseline. First, the model's test F1 score is significantly lower compared to previous experiments, while the train and cross-validation (where the validation set is sampled from the training distribution, disjoint from training data) performance remains comparable to earlier numbers. Such a big drop in F1 score from cross-validation to test is usually a signal of difference in train/test data distribution. We further verify this with a density-based clustering of the train/test data, which results in very small fraction of clusters having data from both train and test sets. Second, with such train/test difference, the model's aggregate F1 score is only marginally higher compared with using the optimizer.

We now evaluate how the models' F1 scores change as we include a tiny fraction of plans from the held-out database into training. The held out setup mimics new applications and databases being deployed in a cloud platform, where execution data accumulates over time after deployment and a new model gets retrained and redeployed.

Figure 9 plots the average F1 score over fifteen databases as k plans for each query in the held-out database are moved from test data to training data. We vary k from 0 which corresponds to using the offline model for an unseen database

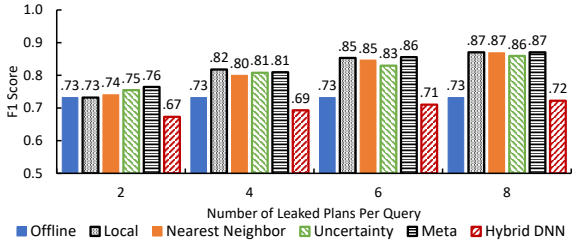


Figure 10: F1 score for adaptive models varying the number of plans leaked per query.

up to 8, at steps of 2. The different bars correspond to two different ways of combining pairs of plans: pair_diff_ratio and pair_diff_normalized. Here, we use *EstNodeCost, LeafWeightEstBytesWeightedSum* feature channels.

With 4 leaked plans, there is a significant increase in the model's F1 score compared to $k = 0$, and the F1 score increases as more plans are added to training. This experiment is further evidence of the differences in train/test distribution and that additional training data from the test distribution helps improve the offline model.

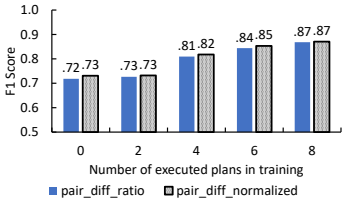


Figure 9: F1 score of the RF model as plans from the held-out database are gradually added to the training set.

7.8 Adaptive Model

Given the need of adaptation, we now evaluate how quickly we can adapt the offline model with new plans available on a database to improve its prediction. Similar to Section 7.7, we hold out the plans in a test database and gradually move plans from test to training.

Figure 10 shows the average F1 score over all the datasets varying the number of plans leaked to training. We evaluate a number of adaptive models discussed in Section 4.3 and 6.2.3, including the local model (*Local*), uncertainty-based adaption (*Uncertainty*), nearest neighbors-based adaption (*Nearest Neighbor*), the meta model (*Meta*), and transfer learning with the hybrid DNN model (*HybridDNN*). We use the probability associated with the prediction in RF to calculate uncertainty as subtracting the probability of the predicted class from 1, and we use cosine distance as the distance metric in nearest neighbors. As more plans are leaked, the performance of all adaptive models improves. Except for hybrid DNN, all other models outperforms the offline model (*Offline*) even with 2 plans leaked. With 8 plans leaked, the best adaptive model achieves 0.87 F1 score. That is, in contrast to *Offline*, *the adaptive models have reduced the fraction of errors by* $\sim 2\times$ *with only* 8 *out of hundreds of plans per query.* In addition, the meta model effectively combines the local and offline models, often beating the local model's F1 score. DNN's performance is inferior compared to RF, likely due to how different the train/test distributions are and all but the last layer's parameters are learned from the train distribution, as noted by Yosinki et al. [70]. All adaptive models train within a minute for most databases, thus allowing them to be retrained on every invocation of the tuner.

7.9 Index Recommendation Quality

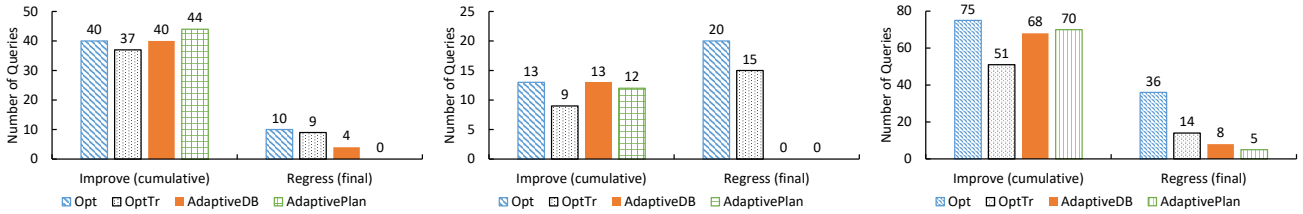
We now evaluate the impact of RF-based classifier integrated into the index tuner on the end-to-end recommendation quality in terms of execution cost (CPU time). We consider the scenario of continuous index tuning as described in Section 2.1, where the number of iterations is 10 and the maximal number of indexes recommended per iteration is 5.

Baselines: We compare with two baselines: (*a*) *Opt*: the original index tuner with optimizer's estimated cost and (*b*) *OptTr*: the index tuner with optimizer's estimated cost where a configuration is only recommended when the estimated improvement is greater than a threshold (set to 20% to match the threshold α used for the classifier).

Tuner with model: Similar to our earlier experiments on models in isolation, we consider two settings of how training data is collected: (*a*) *Split by plan (*AdaptivePlan*):* where the offline model is trained on some plans that are collected from a database before tuning starts in addition to plans from other databases. (*b*) *Split by database (*AdaptiveDB*):* only use data collected from databases *other* than the one being tuned. Since we tune the database continuously, additional indexes are implemented after each iteration and new execution data becomes available. We passively collect the additional execution data and retrain the adaptive component of the adaptive models on every invocation of the tuner.

Since our goal is to prevent regressions, if a query regresses after implementing the index configuration, we *revert* the indexes. Because *Opt* and *OptTr* do not take feedback from previous executions, once a query regresses, they will recommend the same reverted indexes in future iterations and hence cannot make any progress. Thus, for *Opt* and *OptTr*, the tuning stops when there is no index recommendation available or a query regresses. In contrast, as *AdaptivePlan* and *AdaptiveDB* are retrained with additional execution data, after a regression and indexes reverted, it can recommend an alternative configuration in the next iteration. Thus, the tuning stops if only no such configuration is found.

Workloads: We evaluate the end-to-end tuning performance with three workloads: (*a*) *TPC-DS 10g* with no index as initial configuration. (*b*) *TPC-DS 100g* with existing columnstore as



(a) TPC-DS 10g (b) TPC-DS 100g (c) Customer6

Figure 11: Number of queries improved at its final configuration (with regressed configuration reverted) and regressed at the last iteration for query-level tuning with ten iterations.

initial configuration. (*c*) *Customer6* with no index as initial configuration. Customer6 is the most complex real customer workload having many queries with tens of joins. We only tune queries with original CPU cost ≥ 500ms, i.e., expensive queries, resulting in 126 queries out of more than 300 queries.

Query-level tuning: We first evaluate single query tuning, a common scenario where a DBA or the system tunes a single expensive or problematic query. We tune each query separately to get the best index recommendation for a query. Figure 11 summarizes the results with *Opt*, *OptTr*, *AdaptivePlan*, and *AdaptiveDB* on the three workloads. We report two metrics: (*a*) *Improve (cumulative)*: the number of queries that are improved at least by 20% in execution cost at the final configuration compared with that at the initial configuration. If a query regresses when the tuning stops, we revert the recommended indexes and use its execution cost before the regressed tuning iteration as its final execution cost; (*b*) *Regress (final)*: the number of queries that regress when the tuning stops. Note *Improve (cumulative)* and *Regress (final)* are not *exclusive*. For example, a query improves at early iterations and regresses at iteration ten. It can be included in both *Improve (cumulative)* and *Regress (final)* if the query still improves over its initial execution cost after reverting the index recommendation at iteration ten. We further show the breakdown of the distribution of improvement for queries in Table 6 of Appendix A.5.

With *Opt*, the percent of queries that end up with regressions can be as high as 29% (see Figure 11(c)), which aligns with our observation of the percent of regressions using auto-indexing on Azure SQL Database [8]. While *OptTr* only recommends indexes with enough estimated improvement, it fails to prevent many regressions. For example, it reduces only one out of ten regressions in TPC-DS 10g (see *Regress (final)* in Figure 11(a)). This confirms our analysis in Figure 1 that huge estimated improvement can still result in significant regressions. Moreover, *OptTr* can backfire and prevent improvement, since it will stop the tuning process early if there is not enough estimated improvement. For example, *OptTr* reduces 22 regressions while losing 24 improved queries (see Figure 11(c)), including 11 improved queries with ≥ 10× or more improvement (see Table 6).

Table 4: Distribution of workload-level execution cost improvement after ten iterations with Opt, OptTr (Tr), AdpativeDB (ADB), AdaptivePlan (APlan).

Workload	TPC-DS 10g				TPC-DS 100g				Customer6			
Improve	Opt	Tr	ADB	APlan	Opt	Tr	ADB	APlan	Opt	Tr	ADB	APlan
0.2	5	5	5	9	5	5	3	4	8	6	5	9
1	3	3	3	3					5	6	9	8
5			1	1								
10									1			
Total	8	8	9	**13**	**5**	**5**	3	4	14	12	14	**17**

In contrast, *AdaptiveDB* and *AdaptivePlan* consistently and significantly reduce the number of regressed queries when the tuning stops, e.g., reducing *Regress (final)* from 20 to 0 in TPC-DS 100g (see Figure 11(b)). Since the models are not perfect, we can lose improvement, e.g., reduce *Improve (cumulative)* from 75 to 70 with *AdaptivePlan* in Customer6 (see Figure 11(c)). However, tuning with models does not lose any huge improvement, i.e., the number of queries with 10× or more improvement in Table 6. In fact, since tuning with models eliminates configurations that are predicted to regress, which can lead to a different recommendation than that of *Opt*, it can improve more queries (see *Improve (cumulative)* in Figure 11(a)) and improve queries to a larger degree (see the number of queries with ≥ 100× improvement for Customer6 in Table 6). We further compare models' recommendation quality at different iterations in Appendix A.5. Overall, we observe *AdaptivePlan* recommends better indexes than *AdaptiveDB*, especially in early iterations.

Workload-level tuning: We recommend the best indexes for a set of queries in a workload. For each database workload, we construct twenty query workloads by randomly sampling five queries from that workload. The queries in a query workload have uniform weight. As with query-level tuning, *Opt* and *OptTr* can stop early if there is no index recommendation or any query regresses in the query workload.

At each iteration, models are retrained with execution data from previous iterations with this query workload. Similarly, a configuration will revert to that of the previous iteration if a query regresses. The improvement is computed between the execution cost of the workload at the final configuration (with reversion) and that at the initial configuration.

Table 4 shows the distribution of workload-level execution cost improvement. Overall, *Opt* outperforms *OptTr* and improves 27 out of 60 workloads. *AdaptivePlan* improves 34 workloads or 26% more workloads than *Opt* and it improves some workloads to a larger degree, e.g., TPC-DS 10g. However, the models lose a big improvement in Customer6 due to the tuner with models not being able to improve one expensive query in that query workload.

Overhead: Index tuning with optimizer's cost estimate can take a few minutes to a few hours depending on the complexity of the workload. Training the offline model is not in the critical path of tuning and it only takes tens of minutes. Similarly, training data is collected passively and does not add overhead to tuning. The two major overheads when tuning with models are: online model retraining for adaptive model and model inference during tuning. As noted earlier, retraining the adaptive model even on thousands of plans completes within a minute on one or two threads, and inference using the RF-based model takes tens of microseconds. Together, these add less than 10% tuning time on average.

8 RELATED WORK

Database tuning is critical to improve query performance. Several facets of automatic database tuning have been studied, such as configuration parameters [66, 67], data partitioning schemes [4, 22, 55], memory/buffer management [11, 62], optimized index and data structures [41, 46, 48], self-organized index structures [6, 7, 38–40], and cost models [60, 68]. We focus on index tuning for a given workload.

Automatic index tuning has been an active research area for several decades [30, 61]. The problem was first formulated as a search of indexes on a specified workload [17, 65], leveraging extensions to the query optimizer known as the "what-if" API [18]. Several variants of the problem were subsequently studied with different search strategies [13], integrating index tuning with other physical design structures such as partitioning, materialized views, or column-stores [2, 28, 44, 72], formulating it as a continuous tuning problem [12, 56], or modeling robustness of physical design tuning [29]. All these approaches have focused on using the query optimizer's cost estimates for tuning. On the other hand, this paper considers a complementary problem of recommending indexes that improve actual execution cost.

Many regressions are caused by query optimizer errors and improving the optimizer has been an active research area [14–16, 20, 60, 69]. However, as observed in [47, 50], estimation errors in query optimization remains a significant problem. While improvements in query optimization can help index tuning, such improvements are orthogonal to our approach. Our approach works external to the optimizer and can compliment the optimizer unless the optimizer never makes errors. Application of our classification-based cost comparison to improve query optimization is future work.

Recently, there has been a renaissance of applying ML techniques to database tuning problems. Pavlo et al. [52] and Sharma et al. [58] use advanced ML techniques such as neural networks or reinforcement learning to automate index tuning. Since only the vision is presented and details are missing, it is hard to compare these techniques with our approach. We focus on improving recommendation quality in terms of execution cost as well as present techniques to quickly adapt to unseen data, lifting the closed-world assumption of similar train/test distribution of most ML techniques.

9 CONCLUSION

Automated indexing in production systems requires that index recommendations improve query execution cost, not just query optimizer's estimates. We study the problem of leveraging query execution statistics to improve index recommendation quality with minimal changes to state-of-the-art index tuners. We present our key insight that the tuner frequently needs to compare the execution cost of the plans, and learning a classifier that directly minimizes the comparison errors results in significantly higher accuracy in comparing plans in contrast to using learned or analytical regressors. We present an extensive study of the design space of ML techniques, address the issue of different train/test distributions often observed in our setting, and propose an effective lightweight adaptive model that relaxes the closed-world assumption made in most applications of ML in systems. Thorough evaluation using benchmarks and customer workloads demonstrates that learning to compare plans can result in $2\times - 5\times$ reduction in errors. Integrating the model into an index tuner can significantly improve recommendation quality by eliminating most execution cost regressions while preserving the improvement.

It is worth noting that the query optimizer, during its search of a query's plan, also needs to compare sub-plans to find the best sub-plan for a query sub-expression. Many of today's cost-based optimizers use cost estimates for this comparison. Applying our classification-based comparison to query optimization is interesting future work.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers, our shepherd Matthias Boehm, Anshuman Dutt, and Arnd Christian König for their valuable feedback.

REFERENCES

[1] Martín Abadi et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*. 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[2] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollár, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. 2004. Database Tuning Advisor for Microsoft SQL Server 2005. In *VLDB*. 1110–1121.

[3] Sanjay Agrawal, Eric Chu, and Vivek Narasayya. 2006. Automatic physical design tuning: workload as a sequence. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM, 683–694.

[4] Sanjay Agrawal, Vivek R. Narasayya, and Beverly Yang. 2004. Integrating Vertical and Horizontal Partitioning Into Automated Physical Database Design. In *SIGMOD*. 359–370. https://doi.org/10.1145/1007568.1007609

[5] Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B. Zdonik. 2012. Learning-based Query Performance Modeling and Prediction. In *ICDE*. IEEE Computer Society, Washington, DC, USA, 390–401. https://doi.org/10.1109/ICDE.2012.64

[6] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. 2014. H2O: a hands-free adaptive store. In *SIGMOD*. 1103–1114. https://doi.org/10.1145/2588555.2610502

[7] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 2016. Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. In *SIGMOD*. 583–598. https://doi.org/10.1145/2882903.2915231

[8] Azure SQL Database [n. d.]. Azure SQL Database. https://azure.microsoft.com/en-us/services/sql-database/.

[9] Christopher M. Bishop. 2007. *Pattern recognition and machine learning, 5th Edition*. Springer. http://www.worldcat.org/oclc/71008143

[10] Renata Borovica, Ioannis Alagiannis, and Anastasia Ailamaki. 2012. Automated physical designers: what you see is (not) what you get. In *DBTest*. 9. https://doi.org/10.1145/2304510.2304522

[11] Kurt P. Brown, Michael J. Carey, and Miron Livny. 1996. Goal-oriented Buffer Management Revisited. In *SIGMOD*. ACM, New York, NY, USA, 353–364. https://doi.org/10.1145/233269.233351

[12] Nicolas Bruno and Surajit Chaudhuri. 2007. An Online Approach to Physical Design Tuning. In *ICDE*. 826–835. https://doi.org/10.1109/ICDE.2007.367928

[13] Nicolas Bruno and Surajit Chaudhuri. 2007. Physical design refinement: The 'merge-reduce' approach. *ACM Trans. Database Syst.* 32, 4 (2007), 28.

[14] Surajit Chaudhuri. 1998. An Overview of Query Optimization in Relational Systems. In *PODS*. ACM, New York, NY, USA, 34–43. https://doi.org/10.1145/275487.275492

[15] Surajit Chaudhuri. 2009. Query optimizers: time to rethink the contract?. In *SIGMOD*. 961–968. https://doi.org/10.1145/1559845.1559955

[16] Surajit Chaudhuri, Vivek Narasayya, and Ravi Ramamurthy. 2008. A pay-as-you-go framework for query execution feedback. *PVLDB* 1, 1 (2008), 1141–1152.

[17] Surajit Chaudhuri and Vivek R. Narasayya. 1997. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 146–155.

[18] Surajit Chaudhuri and Vivek R. Narasayya. 1998. AutoAdmin 'What-if' Index Analysis Utility. In *SIGMOD*. 367–378. https://doi.org/10.1145/276304.276337

[19] Surajit Chaudhuri and Vivek R. Narasayya. 2007. Self-Tuning Database Systems: A Decade of Progress. In *VLDB*. 3–14.

[20] Chungmin Melvin Chen and Nick Roussopoulos. 1994. Adaptive Selectivity Estimation Using Query Feedback. In *SIGMOD*. ACM, New York, NY, USA, 161–172. https://doi.org/10.1145/191839.191874

[21] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *SIGKDD*. 785–794. https://doi.org/10.1145/2939672.2939785

[22] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. 2010. Schism: a workload-driven approach to database replication and partitioning. *PVLDB* 3, 1-2 (2010), 48–57.

[23] Benoît Dageville, Dinesh Das, Karl Dias, Khaled Yagoub, Mohamed Zaït, and Mohamed Ziauddin. 2004. Automatic SQL Tuning in Oracle 10g. In *VLDB*. 1098–1109. http://www.vldb.org/conf/2004/IND4P2.PDF

[24] Sudipto Das, Miroslav Grbic, Igor Ilic, Isidora Jovandic, Andrija Jovanovic, Vivek Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu, and Surajit Chaudhuri. 2019. Automatically Indexing Millions of Databases in Microsoft Azure SQL Database. In *SIGMOD*. ACM. https://doi.org/10.1145/3299869.3314035

[25] Debabrata Dash, Neoklis Polyzotis, and Anastasia Ailamaki. 2011. CoPhy: A Scalable, Portable, and Interactive Index Advisor for Large Workloads. *PVLDB* 4, 6 (2011), 362–372. https://doi.org/10.14778/1978665.1978668

[26] Bailu Ding, Sudipto Das, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. 2018. Plan Stitch: Harnessing the Best of Many Plans. *PVLDB* 11, 10 (2018), 1123–1136.

[27] Jennie Duggan, Ugur Çetintemel, Olga Papaemmanouil, and Eli Upfal. 2011. Performance prediction for concurrent database workloads. In *SIGMOD*. 337–348. https://doi.org/10.1145/1989323.1989359

[28] Adam Dziedzic, Jingjing Wang, Sudipto Das, Bolin Ding, Vivek R. Narasayya, and Manoj Syamala. 2018. Columnstore and B+ tree - Are Hybrid Physical Designs Important?. In *SIGMOD*. 177–190. https://doi.org/10.1145/3183713.3190660

[29] Kareem El Gebaly and Ashraf Aboulnaga. 2008. Robustness in automatic physical database design. In *EDBT*. 145–156. https://doi.org/10.1145/1353343.1353365

[30] Sheldon J. Finkelstein, Mario Schkolnick, and Paolo Tiberio. 1988. Physical Database Design for Relational Databases. *ACM Trans. Database Syst.* 13, 1 (1988), 91–128. https://doi.org/10.1145/42201.42205

[31] Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L. Wiener, Armando Fox, Michael Jordan, and David Patterson. 2009. Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning. In *ICDE*. IEEE Computer Society, Washington, DC, USA, 592–603. https://doi.org/10.1109/ICDE.2009.130

[32] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[33] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable Feature Learning for Networks. In *SIGKDD*. 855–864. https://doi.org/10.1145/2939672.2939754

[34] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *ICCV*. 1026–1034. https://doi.org/10.1109/ICCV.2015.123

[35] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[36] Kurt Hornik. 1991. Approximation capabilities of multilayer feedforward networks. *Neural Networks* 4, 2 (1991), 251–257. https://doi.org/10.1016/0893-6080(91)90009-T

[37] Kurt Hornik, Maxwell B. Stinchcombe, and Halbert White. 1989. Multilayer feedforward networks are universal approximators. *Neural*

Networks 2, 5 (1989), 359–366. https://doi.org/10.1016/0893-6080(89)90020-8

[38] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database Cracking. In *CIDR*. 68–78.

[39] Stratos Idreos, Stefan Manegold, and Goetz Graefe. 2012. Adaptive indexing in modern database kernels. In *EDBT*. 566–569. https://doi.org/10.1145/2247596.2247667

[40] Stratos Idreos, Stefan Manegold, Harumi A. Kuno, and Goetz Graefe. 2011. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *PVLDB* 4, 9 (2011), 585–597. https://doi.org/10.14778/2002938.2002944

[41] Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S. Kester, and Demi Guo. 2018. The Data Calculator: Data Structure Design and Cost Synthesis from First Principles and Learned Cost Models. In *SIGMOD*. 535–550. https://doi.org/10.1145/3183713.3199671

[42] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In *NIPS*. 3149–3157. http://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree

[43] Keras 2018. Keras: The Python Deep Learning library. https://keras.io/.

[44] Michael S. Kester, Manos Athanassoulis, and Stratos Idreos. 2017. Access Path Selection in Main-Memory Optimized Data Systems: Should I Scan or Should I Probe?. In *SIGMOD*. 715–730. https://doi.org/10.1145/3035918.3064049

[45] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *CoRR* abs/1412.6980 (2014). arXiv:1412.6980 http://arxiv.org/abs/1412.6980

[46] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. 489–504. https://doi.org/10.1145/3183713.3196909

[47] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9, 3 (Nov. 2015), 204–215. https://doi.org/10.14778/2850583.2850594

[48] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*. 38–49. https://doi.org/10.1109/ICDE.2013.6544812

[49] Jiexing Li, Arnd Christian König, Vivek R. Narasayya, and Surajit Chaudhuri. 2012. Robust Estimation of Resource Consumption for SQL Queries using Statistical Techniques. *PVLDB* 5, 11 (2012), 1555–1566. https://doi.org/10.14778/2350229.2350269

[50] Guy Lohman. 2014. Is Query Optimization a "Solved" Problem? http://wp.sigmod.org/?p=1075.

[51] ML.NET 2018. Machine Learning for .NET. https://github.com/dotnet/machinelearning.

[52] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C. Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. 2017. Self-Driving Database Management Systems. In *CIDR*.

[53] Lorien Y. Pratt. 1992. Discriminability-Based Transfer between Neural Networks. In *NIPS*. 204–211. http://papers.nips.cc/paper/641-discriminability-based-transfer-between-neural-networks

[54] Program for TPC-H Data Generation with Skew [n. d.]. Program for TPC-H Data Generation with Skew. https://www.microsoft.com/en-us/download/details.aspx?id=52430.

[55] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy M. Lohman. 2002. Automating physical database design in a parallel database. In *SIGMOD*. 558–569. https://doi.org/10.1145/564691.564757

[56] Karl Schnaitter, Serge Abiteboul, Tova Milo, and Neoklis Polyzotis. 2006. COLT: continuous on-line tuning. In *SIGMOD*. 793–795. https://doi.org/10.1145/1142473.1142592

[57] scikit-learn 2018. scikit-learn: Machine Learning in Python. http://scikit-learn.org/stable/.

[58] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. 2018. The Case for Automatic Database Administration using Deep Reinforcement Learning. *CoRR* abs/1801.05643 (2018).

[59] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. 2015. Training Very Deep Networks. In *NIPS*. 2377–2385. http://papers.nips.cc/paper/5850-training-very-deep-networks

[60] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO - DB2's LEarning Optimizer. In *VLDB*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 19–28.

[61] Michael Stonebraker. 1974. The choice of partial inversions and combined indices. *International Journal of Parallel Programming* 3, 2 (1974), 167–188. https://doi.org/10.1007/BF00976642

[62] Adam J Storm, Christian Garcia-Arellano, Sam S Lightstone, Yixin Diao, and Maheswaran Surendra. 2006. Adaptive self-tuning memory in DB2. In *VLDB*. VLDB Endowment, 1081–1092.

[63] TPC Benchmark DS: Standard Specification v2.6.0. [n. d.]. TPC Benchmark DS: Standard Specification v2.6.0. http://www.tpc.org/tpcds/.

[64] TPC Benchmark H: Standard Specification v2.17.3. [n. d.]. TPC Benchmark H: Standard Specification v2.17.3. http://www.tpc.org/tpch/default.asp.

[65] 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. https://doi.org/10.1109/ICDE.2000.839397

[66] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *SIGMOD*. ACM, New York, NY, USA, 1009–1024. https://doi.org/10.1145/3035918.3064029

[67] Gerhard Weikum, Axel Moenkeberg, Christof Hasse, and Peter Zabback. 2002. Self-tuning database technology and information services: from wishful thinking to viable engineering. In *VLDB*. Elsevier, 20–31.

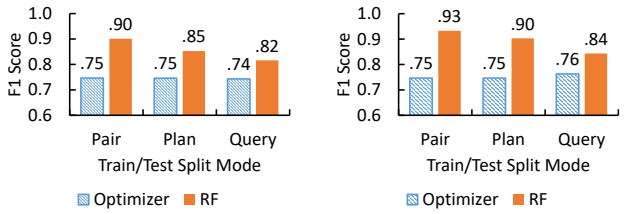
[68] Wentao Wu, Yun Chi, Shenghuo Zhu, Jun'ichi Tatemura, Hakan Hacigümüs, and Jeffrey F. Naughton. 2013. Predicting Query Execution Time: Are Optimizer Cost Models Really Unusable?. In *ICDE*. IEEE Computer Society, Washington, DC, USA, 1081–1092. https://doi.org/10.1109/ICDE.2013.6544899

[69] Wentao Wu, Jeffrey F. Naughton, and Harneet Singh. 2016. Sampling-Based Query Re-Optimization. In *SIGMOD*. ACM, New York, NY, USA, 1721–1736. https://doi.org/10.1145/2882903.2882914

[70] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. 2014. How transferable are features in deep neural networks?. In *NIPS*. 3320–3328. http://papers.nips.cc/paper/5347-how-transferable-are-features-in-deep-neural-networks

[71] Haijun Zhang, Shuang Wang, Xiaofei Xu, Tommy W. S. Chow, and Q. M. Jonathan Wu. 2018. Tree2Vector: Learning a Vectorial Representation for Tree-Structured Data. *IEEE Trans. Neural Netw. Learning Syst.* 29, 11 (2018), 5304–5318. https://doi.org/10.1109/TNNLS.2018.2797060

[72] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy Lohman, Adam Storm, Christian Garcia-Arellano, and Scott Fadden. 2004. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *VLDB*. VLDB Endowment, 1087–1097.



(a) Train with 10% data (b) Train with 50% data

Figure 12: F1 score when training and testing on plans obtained from a production auto-indexing service with different fractions of data used in training.

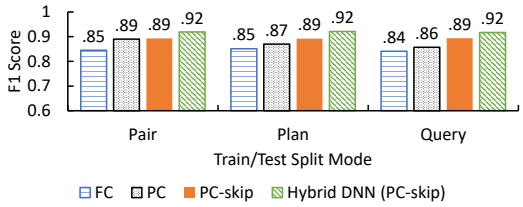


Figure 13: F1 score for different DNN architectures.

A ADDITIONAL EXPERIMENTS

A.1 Classification on production data

We collect anonymized and aggregated execution data from a production auto-indexing service in Azure SQL Database [24]. This service continuously analyzes databases and workloads to recommend indexes that can improve query execution cost. For databases where indexes are automatically implemented, the service collects query execution data before and after the change. These executions are on a production database running queries concurrently, in contrast to the controlled setting of execution data collection in Section 7. This production data includes various statement types, e.g., `SELECT`, `UPDATE`, `INSERT`, `DELETE`, `UPSERT`, in addition to the query and data diversity. We collected two weeks of execution data from $\sim 10K$ databases with $\sim 40K$ plans, resulting in $\sim 750K$ plan pairs. For many queries, we observed tens of plans, and more than 80% queries have at least 2 plans.

We evaluate the model on this production data. Similar to the experiment in Section 7.6, we split the plan pairs into train/test using three modes: Pair, Plan, and Query. We vary the fraction of data in the training set from 0.1 - 0.5, where the remainder of the data is in the test set. We compare the best classifier model (i.e., RF) with using the optimizer's estimates. Figure 12 reports the F1 score for the different split modes with train/test ratio 0.1 and 0.5. Even with 0.1 train ratio, the classifier's F1 score is significantly higher compared to the optimizer, with performance being higher when train and test data distributions are similar (e.g., split by pairs).

Table 5: F1 score for RF when holding out all execution statistics from a database, with different feature channels and ways to combine features from plan pair.

	NC		NC, LWB		NC, LWR		NC, LWB, BP		NC, LWR, RP	
	Avg.	Median	Avg.	Median	Avg.	Median	Avg.	Median	Avg.	Median
pair_diff	0.7	0.73	0.71	0.75	0.7	0.73	0.7	0.75	0.7	0.74
pair_diff_ratio	0.7	0.75	0.71	0.77	0.71	0.77	0.72	0.76	0.71	0.76
pair_diff_normalized	0.72	0.75	0.72	0.75	0.72	0.75	0.73	0.75	0.74	0.75

This experiment shows the effectiveness of the classifier even with diverse workloads, concurrently executed queries, and various statement types from thousands of databases in production systems.

A.2 Regression vs. Classification

In Section 7.5, we evaluate and compare the F1 score for regression prediction using the classifier, the regressors, and the optimizer. We further compute the workload cost using the models for plan comparison.

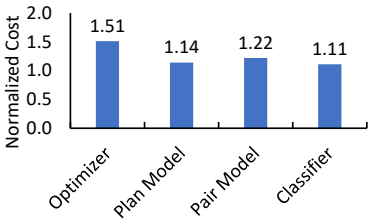


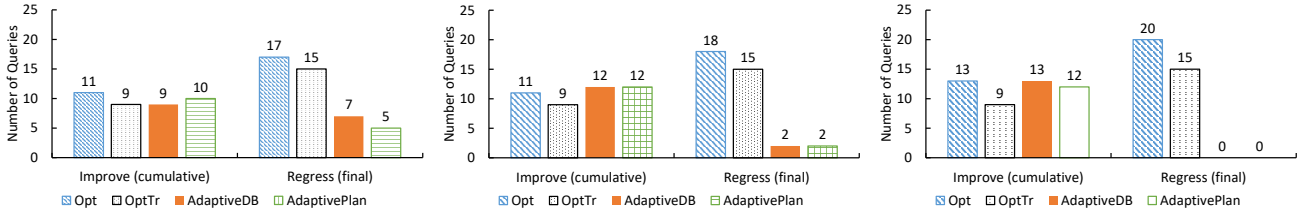
Figure 15: Normalized workload cost with different models.

For each plan pair $\mathcal{P}_1, \mathcal{P}_2$, if the model predicts a regression, we select plan $\mathcal{P}_1$; otherwise, we select plan $\mathcal{P}_2$. For each model, we sum up the execution cost of all the plans selected as the *workload cost* using the model. The sum of the execution cost of the cheaper plan in each pair is the *optimal workload cost.*

Figure 15 shows the workload cost for all the models in Section 7.5 normalized by the optimal workload cost. As is evident, the classifier outperforms all the other models, with the optimizer being the worst. This further confirms our key insight that formulating the regression prediction as a classification task rather than a regression task is more appropriate for index recommendation.

A.3 Feature sensitivity

In Section 7.7, we observe the need of adaptation for unseen databases where train/test distributions are significantly different. To ensure that such differences in train/test distributions are not due to a specific choice of feature channels or how we combine the vectors for a plan pair into the final vector, we repeat the experiment (as in Section 7.7) for different featurization: (*i*) by varying feature channels (Section 3.2); and (*ii*) by varying how we combine vectors (Section 3.3). We pick different subsets of channels from Table 1. We report



(a) Iteration 1 (b) Iteration 3 (c) Iteration 10

Figure 14: Number of queries improved at the current configuration (with regressed configuration reverted) and regressed at the current iteration during tuning at various iterations for workload TPC-DS 100g.

Table 6: Distribution of execution cost improvement after ten iterations for query-level continuous tuning.

Workload	TPC-DS 10g				TPC-DS 100g				Customer6			
Improve	Opt	Tr	ADB	APlan	Opt	Tr	ADB	APlan	Opt	Tr	ADB	APlan
0.2	12	10	13	13	3	2	3	1	7	8	10	10
1	16	15	14	20	6	3	5	5	30	19	21	16
5	6	6	7	5	2	2	1	1	8	5	5	9
10	3	3	3	3	2	2	**4**	3	**21**	13	20	**21**
100	3	3	3	3	0	0	0	**2**	9	6	12	**14**
Total	40	37	40	44	13	9	13	12	75	51	68	70

the F1 score for different featurization in Table 5. Each pair of columns corresponds to a subset of feature channels, we report the average and median of fifteen runs where each of the fifteen databases is held out from training and used in testing. In the table, NC is for *EstNodeCost*, LWB is for *LeafWeightEstBytesWeightedSum*, LWR is for *LeafWeightEstRowsWeightedSum*, BP is for *EstBytesProcessed*, RP is for *EstRowsProcessed*. While there are minor differences in F1 score across various featurization, the RF's F1 score remains significantly lower compared to that reported in Figure 7. Hence, the train/test differences hold across various featurization. We also observe that an appropriate choice of featurization has an impact on the model's generalization to unseen data.

A.4 DNN Architectures

In Section 6.2.1, we present several novel DNN architectures, gradually increasing the sophistication starting from the fully-connected network architecture. Here we evaluate the impact of those novel architectural changes on the model's predictions. Figure 13 plots the F1 score for the different DNN architectures: (*a*) fully-connected (FC); (*b*) partially-connected (PC) with 3 PC layers; (*c*) partially-connected with skip connections (connecting the output of layer $\mathcal{L}_i$ to input of $\mathcal{L}_{i+2}$), i.e., PC-skip; and (*d*) Hybrid DNN. We split train/test in different ways, with 60% used for training and 40% for testing. The DNN in each setting has 12 hidden layers with 64 neurons each, with other hyper-parameters set to values specified in Section 7.4. The Hybrid DNN uses the PC-skip DNN and adds an RF with 50 estimators. This experiment shows the incremental improvement from different architectures with an approximately 10 percentage point increase in F1 score of a more sophisticated design (*d*) compared to the simple FC network architecture (*a*).

A.5 Index recommendation

In Section 7.9, we show the summary of the query-level tuning result. Table 6 breaks down the result and shows the distribution of the improvement for query-level tuning. As is evident from the table, both *AdaptiveDB* (ADB) and *AdaptivePlan* (APlan) retain similar or more queries with 10× or more improvement as compared with *Opt*, while *OptTr* significantly reduces the number of such improved queries.

In Section 7.8, we show that the F1 score plunges when we train the model with unseen database compared to that when the training data includes the execution data from the target database, and with adaptive models and four plans per query leaked from the unseen database, the F1 score improves from 0.73 to 0.82. Thus, with continuous tuning, the adaptive models learn from the passively collected execution data and can potentially improve index recommendation quality after a few iterations compared to the offline model.

Figure 14 shows the number of improved queries with regressed configuration reverted (*Improve (cumulative)*) and the number of regressed queries (*Regress (final)*) on TPC-DS 100g at various iterations. At iteration 1, *AdaptivePlan* outperforms *AdaptiveDB* because initially *AdaptivePlan* is trained with data from other databases and TPC-DS 100g while *AdaptiveDB* is only trained with data from other databases. As the tuning continues, *AdaptiveDB* is retrained with passively collected data from previous iterations and quickly catches up with *AdaptivePlan* at iteration 3, eventually resulting in no regressed queries and similar number of improved queries with *AdaptivePlan* at iteration 10. Thus, the adaptive models effectively improve prediction performance as more execution data becomes available.