T3: Accurate and Fast Performance Prediction for Relational Database Systems With Compiled Decision Trees

Maximilian Rieger
Technische Universität München
Munich, Germany
max.rieger@tum.de

Thomas Neumann
Technische Universität München
Munich, Germany
neumann@in.tum.de

Abstract

Query performance prediction is used for scheduling, resource scaling, tenant placement, and various other use-cases. Here, the main goal is to estimate the execution time of a query without running it. To be effective, predictors need to be both accurate and fast. In contrast, neural networks that were used in recent work deliver very accurate predictions but suffer from high latency.

In this work, we propose the **T**uple **T**ime **T**ree (T3), a new model that is both accurate and fast. It is orders of magnitude faster than comparable methods and has competitive accuracy to state-of-the-art approaches. Additionally, T3 works for new database instances without re-training because it generalizes across database instances.

We achieve T3's speed by relying on a low-latency decision tree model that is compiled to native machine code. We maintain high accuracy with two novel techniques: pipeline-based query plan representation and tuple-centric prediction targets.

In our pipeline-based query plan representation, T3 decomposes query plans into pipelines. Then, T3 predicts the execution time of each pipeline individually, instead of the whole query in one step. With tuple-centric prediction targets, T3 predicts the expected time it takes to push a single tuple through a pipeline. It then multiplies this predicted value by the input cardinality of the pipeline to estimate its execution time. As a result, T3 achieves state-of-the-art accuracy with a low-latency decision tree model.

CCS Concepts

• **Information systems** → **Database query processing**.

Keywords

Database Systems, Query Performance Prediction, Cost Model

ACM Reference Format:
Maximilian Rieger and Thomas Neumann. 2025. T3: Accurate and Fast Performance Prediction for Relational Database Systems With Compiled Decision Trees. In *Proceedings of the 2025 International Conference on Management of Data (SIGMOD'25).* ACM, New York, NY, USA, Article 227, 14 pages. https://doi.org/10.1145/3725364

Authors' Contact Information: Maximilian Rieger, Technische Universität München, Munich, Germany, max.rieger@tum.de; Thomas Neumann, Technische Universität München, Munich, Germany, neumann@in.tum.de.

SIGMOD'25, June 22–27, 2025, Berlin, Germany

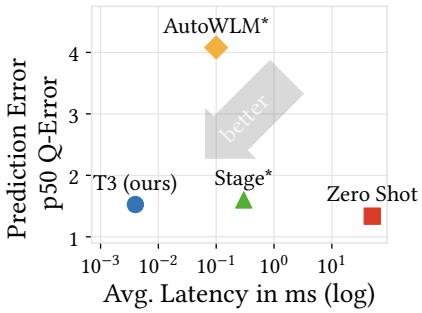

Figure 1: Latency and accuracy of recent models [16, 40, 50]. (*values from [50])

1 Introduction

Motivational Example. Modern cloud-based data systems are designed to dynamically scale resources and automate query scheduling to handle diverse workloads. These systems often face significant spikes in concurrent query submissions, requiring efficient scheduling across multiple compute clusters to ensure good performance and resource utilization. Scheduling algorithms assign queries based on predicted resource requirements. Better predictions can significantly improve overall system performance. For instance, Amazon Redshift has demonstrated notable performance gains by integrating an improved performance prediction model [50]. Yet, substantial potential for further improvement remains. Wu et al. [50] suggest that average query performance could still improve by over 40% with more accurate models. Another area for improvement is prediction latency. Since each query must wait for its prediction before being scheduled, this latency directly adds to the total execution time of every query, which significantly impacts short running queries. This is a scenario where improvements in both model accuracy and latency can substantially enhance modern data management systems.

Performance Prediction Goals. We identify three main goals for performance prediction. First, predictions should be as accurate as possible. Second, predictions should be made with low latency. And finally, prediction models should work on new database instances without any new training data.

Use-Cases. Modern data management systems utilize query performance prediction models for various use-cases. These include scheduling [9, 11, 12, 39, 50], data encoding selection [4], admission control [45], cloud resource scaling [8, 27], tenant placement [44], holistic workload management [31, 40], materialized view

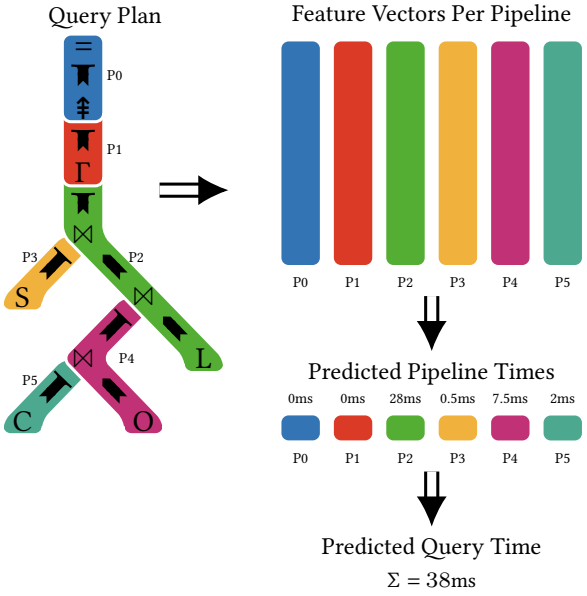

Figure 2: T3 prediction process for TPC-H Q5. Each pipeline is converted to a flat feature vector. T3's decision tree model predicts a time for each feature vector. The sum of the predicted pipeline times is the result.

creation [40], cloud architecture design [54], and query optimization [2, 16, 29, 43, 57].

Learned Models. Predicting performance accurately is a long-standing challenge that was initially addressed using hand-crafted, detailed models of database systems [25, 28]. In contrast, recent learned models capture system performance characteristics automatically and yield significantly better accuracy [16, 43, 50]. Many of these learned methods use neural networks, valued for their flexibility in input encoding and accuracy in approximating complex nonlinear functions. However, they come at the cost of high prediction latency, which is crucial for many use-cases [40, 50].

T3. In this work, we propose the **T**uple **T**ime **T**ree (T3), a compiled decision tree model for performance prediction. T3 improves prediction latency by orders of magnitude while maintaining state-of-the-art-accuracy (see Figure 1). The following paragraphs summarize its main novelties and features.

Prediction Latency. With about 4us inference latency, T3 is roughly four orders of magnitude faster than neural network models such as Zero Shot [16]. Even compared to models that use decision trees like AutoWLM, T3 is orders of magnitude faster (see Figure 1). We achieve this speedup by compiling our model to native machine code [3].

Pipeline-Based Query Plan Representation. One of T3's main novelties is its pipeline-based query plan representation. Query plans can be decomposed into multiple pipelines, which scan some input and fully materialize their result. We apply T3's decision tree to predict the execution time of each pipeline individually. Then, we obtain the execution time of the whole query by summation of the execution times of all its pipelines. This decomposition simplifies the performance prediction problem and improves accuracy for decision trees significantly. See Section 2.2 for further details.

Tuple-Centric Prediction Targets. We predict the expected running time of pushing a single tuple into a pipeline instead of directly predicting the execution time of that pipeline for all tuples. We also build our feature vectors with tuple-centric predictions in mind. To estimate the whole execution time of a pipeline, we multiply the prediction value by the input cardinality of the pipeline. This fits the architecture of decision trees much better and improves prediction accuracy substantially. See Section 2.4 for further details.

Generalization. Finally, T3 employs a *zero shot* approach as introduced by Hilprecht and Binnig [15, 16]. This means that it is trained on a variety of database instances and generalizes to new ones. Many previous approaches need workload traces and re-training for every new database instance, whereas T3 can be applied instantly without any expensive work.

Contributions. In summary, we make the following contributions:

- We demonstrate low prediction latency by compiling decision tree models to native machine code.
- We present a novel tuple-centric prediction strategy and pipeline-based query plan representations and corresponding feature vectors to achieve state-of-the-art accuracy with decision trees.
- We implement these approaches in T3, an extremely fast performance prediction model with state-of-the-art accuracy and publish the source code and model. [1]
- We thoroughly evaluate T3 with respect to latency and accuracy.

2 The T3 Model

This section describes the key components of T3. First, it covers the exact scope of the prediction problem. Next, it introduces our pipeline-based query plan representation. Then, it covers the architecture of decision trees. Based on this, it explains the exact inputs and outputs of the T3 model. Finally, we show how model compilation improves T3's performance.

2.1 Problem Scope

Performance prediction is the task of predicting how long the database system will take to execute a query without running it. T3 predicts the execution time of Umbra, a flash-based compiling relational database system with in memory performance [36]. T3 works for a wide range of queries including all TPC-DS benchmark queries. To enable accurate predictions that account for hardware characteristics, we build our model for fixed hardware. T3 can be transferred to predict for new hardware by re-running all benchmark queries (few hours) and re-training (few seconds) with the new data. Further, T3 relies on physical query plans for detailed information about queries. A physical query plan includes annotations of expressions, tuple sizes in bytes, and cardinalities of base relations and intermediate results. We intentionally decouple this problem from cardinality estimation. Cardinality estimates will inevitably be bad for complex queries [23]. Instead of trying to improve cardinality estimates, we build a model that works well with

[1] https://github.com/MaxRieger96/T3

good estimates. However, it can also benefit from ever-improving cardinality estimation methods [18, 53]. In this work, we focus on performance prediction for Umbra, but the concepts are transferable to all kinds of database systems.

2.2 Pipeline-Based Query Plan Representation

Decision Trees Require Flat Feature Vectors. We choose to apply decision trees because of their speed, but they have the limitation that they require a flat feature vector as input. Consequently, previous approaches like AutoWLM represent a query by a single feature vector [40]. However, methods that represent queries with a single vector fail to achieve competitive accuracy.

Query Plans Consist of Pipelines. Most modern database systems only fully materialize intermediate results when absolutely necessary. For example, aggregates or the build-side of hash joins are pipeline breakers that need full materialization. We refer to the path between two such pipeline breakers as a pipeline. A query plan can be viewed as a set of individually executed pipelines, which scan some input, perform computations on it, and materialize their result. See Figure 2 for an example.

Performance Prediction for Pipelines. Our novel approach for better accuracy with decision trees is to predict the execution time for each pipeline individually instead of the whole query. To do so, we encode each pipeline as a single flat feature vector. We then apply T3's decision tree model to each feature vector to predict its execution time. To predict the execution time of the whole query, we sum the execution times of all pipelines. Intuitively, this decomposition of queries into pipelines simplifies the performance prediction problem. Capturing the distinct computational aspects of multiple pipelines within a single prediction is intricate. Predicting for each operator individually, however, introduces new problems, such as benchmarking operators in isolation and accounting for dependencies between consecutive operators. In contrast, predicting for individual pipelines strikes a good balance and conceptually resembles estimating the computational load of a for-loop. Section 5.7 shows that this indeed yields better accuracy. Finally, the use of flat feature vectors enables the use of many types of machine learning models that require flat feature vectors as input. Some operators within a pipeline can exhibit nonlinear effects, such as cache behavior from differently sized hash tables or nonlinear time complexities in sort operators. Hence we need a model capable of capturing these nonlinearities, such as a decision tree.

2.3 Gradient Boosted Trees

We use a gradient boosted tree model from the LightGBM framework [19] due to its high performance and compatibility with the lleaves compilation framework [3]. This model consists of an ensemble of decision trees. Each internal tree node contains a condition and respective branches for the condition's outcomes (see Figure 3). For example, a condition could be $\mathbf{v}^{[1]} < 256$ where $\mathbf{v}^{[1]}$ is the value at index 1 in the feature vector. Starting at the root, we traverse the tree based on the evaluation of these conditions on the input. Finally, we end up at a leaf that holds a prediction value. For example, for the vector $\mathbf{v} = (1, 384, 64)^\intercal$, we predict the value 8ms. To improve the accuracy of such decision trees, gradient boosting

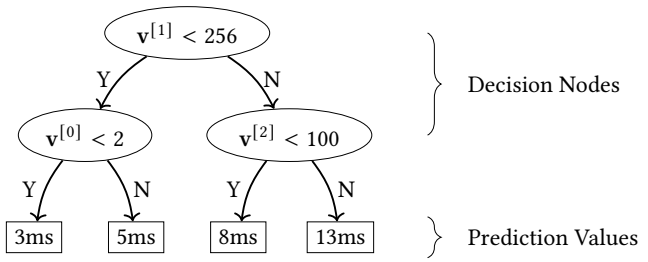

Figure 3: Example of a two level decision tree. During evaluation, we traverse the decision nodes of the tree. Once we reach a leaf, we return the corresponding prediction value.

methods iteratively build new trees. Each new tree aims to predict the residual error of predictions of all previous trees. Ideally, predictions become more accurate with each added tree until they reach a point of diminishing returns. The sum of all values in the leaves forms the model prediction. In our case, we use 200 trees with roughly 30 leaves each.

2.4 Model Input and Output

Tuple-Centric Prediction. Because the possible prediction values of decision trees are limited by the number of leaves, they can only predict a finite number of values. To enable fine-grained execution time prediction, we predict the time for each tuple in a pipeline instead of the time it takes to process all tuples in the pipeline. Intuitively, the model predicts the expected time the query engine spends pushing one tuple into the pipeline. The product of the predicted value and the number of scanned tuples at the beginning of a pipeline forms the full execution time. Therefore, the model predictions can scale to arbitrary input cardinalities with a limited amount of possible prediction values. In Section 5.7 we show that this results in better accuracy.

Prediction Target Transformation. Since we built our model for a fast system, the prediction targets can be very small. In our dataset, the execution times range from 10^{-15}s to 1s per tuple. Training algorithms for decision trees can optimize different cost functions. To train an accurate model for this wide range of values, we ideally want to minimize a relative instead of absolute error. Otherwise, the absolute error of a single long running query could outweigh the prediction errors of thousands of short running queries. We solve this problem by transforming the prediction targets instead of choosing a specific cost function:

$$t' = -\log(t) \tag{1}$$

where t is the per tuple execution time. The new target values t' are now usually in the range of 0 to 15. Further, relative deviations for very short queries now have the same distance as for longer running ones. We observed significantly improved accuracy predicting for these transformed targets. After this transformation, all loss functions provided by LightGBM yield better accuracy.

2.5 Model Training

LightGBM provides an easy interface to train decision tree models [19]. First we need a dataset for training. To form this dataset,

Table 1: Latencies of different performance prediction models. Decision trees (DT) are faster than neural networks (NN). A hierarchy of different models can improve average latencies. Compiled T3 is the fastest model.

	Cache	DT	NN	Avg
Zero Shot [16]	-	-	50ms	50ms
Stage [50]	~2us	~1ms	~30ms	~300us
T3 interpreted	-	22us	-	22us
T3 (ours)	-	4us	-	**4us**

we convert all pipelines from all queries of all but the test database instance (TPC-DS) to feature vectors and use their respective execution times as targets. Next, we use LightGBM's feature to automatically sample 20% of training data as validation data. To train, we call the update function 200 times using the MAPE objective function. This will create a single gradient boosted model consisting of 200 trees.

2.6 Better Latency With Compilation

Good Prediction Latency. Low latency is a key design goal for cost models [40, 50]. Use-cases like scheduling and other optimization algorithms can significantly delay query execution if they have to wait long for model evaluation. Amazon's Stage model approaches this problem by building a hierarchy of predictors with varying latencies [50]. It uses a fast cache for seen queries, a decision tree model for simple queries, and a neural network model that is the most accurate. Table 1 shows that, while the cache is very fast and the decision trees are still acceptable, the neural network model is very slow. While achieving a significantly improved average latency, Stage still suffers from high latencies whenever the neural network model is employed. In contrast, T3 is much faster because it uses a simpler model and employs compilation to native machine code to speed up the decision tree model.

Model Compilation. A technique that improves latency substantially is model compilation (see Table 1). Instead of using LightGBM to evaluate our model, we compile it to native machine code using the framework *lleaves* [3]. LightGBM evaluates its trees using an interpretation-based approach. In contrast, lleaves compiles internal nodes of a decision tree to only two instructions: One instruction to compute the condition and one branch instruction to jump to the corresponding next node. Leaves are compiled to single return instructions. For instance, the tree in Figure 3 would be compiled to three comparison and three branch instructions as well as four return instructions. lleaves leverages the LLVM compiler framework to convert these instructions to native machine code. The final result is a library providing a single predict function that any program can link against. Note that this compilation has to be performed only once after training and does not add to the inference latency. The compiled version of our model reduces the latency from 22us to about 4us for the average query we observed. This is orders of magnitude faster than any current competitive cost model to the best of our knowledge.

Throughput is Better for Batches. An interesting finding is that throughput performance differs drastically from latency. Evaluating models for a large batch (> 1000) of data points at once is much faster than back-to-back evaluation of single ones. For example, the throughput for neural networks improved by over 1000x in our experiments (see Table 2). However, not all use-cases can benefit from batched processing, so we focus on single query prediction latency in this work.

Table 2: Throughput of models in queries per second.

	Back-to-back	Batched
Neural Network	20	50,000
Decision Tree	45,000	700,000
Compiled Decision Tree	250,000	4,000,000

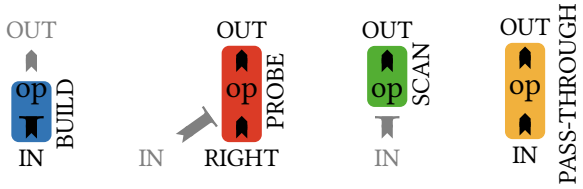

Figure 4: Operators can have multiple stages: build, probe, scan, and pass-through. Furthermore, each operator has up to three tuple streams for data input (IN, RIGHT) or output (OUT). At each end we can consider basic features of the pipeline.

3 Feature Computation

Feature Crafting. We summarize each pipeline in a single feature vector that is used by the model (see Figure 2). These feature vectors need to contain all necessary information for the model to predict the execution time. This section outlines how we form such feature vectors.

Operator Stages. First, we distinguish between different stages of operators as depicted in Figure 4:

- **Build:** Tuples come into the operator and materialize some data. E.g., materialize, aggregation, join hash table build.
- **Probe:** Tuples come from a second (right) input source, perform some computation and leave the operator again. E.g., join probe.
- **Scan:** The operator scans some tuples and outputs them. E.g., table scan, materialize scan, aggregation scan.
- **Pass-Through:** Tuples enter the operator and leave it again. E.g., filter or map.

For example, a join operator has a build stage, that builds a hash table on its left input. But it also has a probe stage, where it probes tuples from the right input side against the hash table and outputs them to the next operator. In contrast, a select operator will only have a pass-through stage where tuples from the input go directly to the output. Finally, a hash-based group-by operator will have a build stage to group elements and a scan stage to output the results.

Tuple Streams. For each operator stage, we consider the relevant tuple streams: IN, OUT, and RIGHT. We use IN for the input of operators that only have one input and for the left input of operators that have two input streams. OUT always denotes the outgoing

```
fn getFeatureVectors(queryPlan):
  vectors = []
  for pipeline in plan.pipelines():
    currentVec = zeros(n_features)
    for op in pipeline.operators():
      stage = op.getStage(pipeline)
      currentVec[countLocation(op, stage)]++
      features = op.getBasicFeatures(stage)
      for feature in features:
        i = getLocation(feature, op, stage)
        if i >= 0:
          currentVec[i] += feature.value
    vectors.append(currentVec)
  return vectors
```

Listing 1: Feature computation for a whole query plan in pseudocode.

stream of an operator. RIGHT refers to the right input of operators that have two input streams. For example, the build stage of a join operator only uses the IN stream. The probe phase involves both RIGHT and OUT.

Basic Features. Computing features in a structured way makes it easily manageable to support all 19 different physical operators and 28 stages that we encountered in our benchmarks. Hence, we use generic features that can be used for different operators, stages, and input/output streams:

- **Percentage** of all tuples at the start of a pipeline that reach a tuple stream.
- **Size** in bytes of single tuples that reach a tuple stream.
- **Cardinality** of an input or output stream.

Our most used feature is percentage. It indicates the fraction of tuples in the pipeline that reach a certain stage. In other terms, it is the product of the selectivities of all its preceding operators. Note the subtle difference to selectivity. This is a very relevant feature, as tuples that do not reach an operator have no effect on its execution performance. E.g., building a hash table for a join is much faster if only 1% of tuples actually reach the operator. We use this feature for all operators. Next, we use the size of each tuple for materializing operators to capture the overhead of storing a tuple. Finally, we use the cardinality of tuple streams to capture nonlinear execution times for materializing operators. For example, we use it for the size of hash tables in joins as well as for the sort operator. In summary, there is a small set of hand-selected features for all possible stages of each operator type. Note that all of these features are tuple-centric, which means they are suitable to predict the expected execution time of pushing a single tuple through the pipeline.

Feature Encoding. Listing 1 describes the process of creating the feature vectors for a query. First, we split the respective query plan into its pipelines. Then, we initialize a new feature vector with only zeros for each pipeline. Note that the size is always fixed to the total number of distinct features `n_features`, in our implementation that is 110. This number is determined by the features added to the individual operator stages and will change when new operators or features are added. Next, we iterate over all operator of the current pipeline. For the operator, we determine, which stage it executes in the current pipeline. Using the respective location, we increment the count of this operator stage in the feature vector. Then we compute all basic features for the operator stage. For each of those features, we determine the corresponding location within the feature vector. Note that we use different subsets of the basic features for different kinds of operator stages. After scanning all operators in the current pipeline, we add the current feature vector to the result collection. Finally, we return the collection of feature vectors that can be used by T3.

```
-- TPC-H Query 5
select
    n_name,
    sum(l_extendedprice * (1 - l_discount))
     as revenue
from
    customer,
    orders,
    lineitem,
    supplier,
    nation,
    region
where
    c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and l_suppkey = s_suppkey
    and c_nationkey = s_nationkey
    and s_nationkey = n_nationkey
    and n_regionkey = r_regionkey
    and r_name = 'ASIA'
    and o_orderdate >= date '1994-01-01'
    and o_orderdate < date '1994-01-01'
        + interval '1' year
group by
    n_name
order by
    revenue desc
```

Listing 2: TPC-H Q5 SQL. This query computes the revenue of all Asian countries in the year 1994 and orders them by revenue.

Feature Encoding Example. As a running example, we use TPC-H query 5 with scale factor 10 (see Listing 2). The respective query execution tree is depicted in Figure 2. Note that Umbra does not include the tables *nation* and *region* in the query plan. Because these tables are very small, the system performs all computation on this data during optimization. Umbra computes all qualifying nation keys and removes the joins with those tables. Instead, it uses *in-expressions* that check whether the respective nation keys are in a list of qualifying values. Listing 3 shows the respective feature vector for pipeline 5. Pipeline 5 scans the *customer* table, selects all tuples that have a qualifying nation key, and builds a hash table over the data. Note that we omitted feature indices and all values that remain zero for legibility. The whole feature vector has over

```
Pipeline5 (scan: 1'500'000 tuples)
  TableScan_Scan_count: 1
  TableScan_Scan_in_card: 1'500'000
  TableScan_Scan_out_percentage: 20%
  TableScan_Scan_in_expression_percentage: 60%
  TableScan_Scan_between_percentage: 100%
  HashJoin_Build_count: 1
  HashJoin_Build_in_card: 300'270
  HashJoin_Build_in_size: 8
  HashJoin_Build_in_percentage: 20%
```

Listing 3: Feature vector of TPC-H Q5 Pipeline 5. All values that are zero are omitted.

```
Pipeline2 (scan: 59'986'052 tuples)
  TableScan_Scan_count: 1
  TableScan_Scan_in_card: 59'986'052
  TableScan_Scan_out_percentage: 100%
  HashJoin_Probe_count: 2
  HashJoin_Probe_in_card: 556'771
  HashJoin_Probe_right_percentage: 103%
  HashJoin_Probe_out_percentage: 3.1%
  GroupBy_Build_count: 1
  GroupBy_Build_out_card: 5
  GroupBy_Build_out_size: 32
  GroupBy_Build_in_percentage: 0.1%
```

Listing 4: Feature vector of TPC-H Q5 Pipeline 2. There are two hash join probe stages in this pipeline. All values that are zero are omitted.

100 elements. We can see that there is one table scan scan stage and one hash join build stage. The input cardinality of the table scan operator is the size of the *customer* table. Out of these 1.5 million tuples, about 20% get selected by expressions. We will discuss these expressions in the 'Table Scan Operators' paragraph later in this section. Consequently, the hash join is also reached by 20% of the input tuples and has a cardinality of about 300k. As the hash table only needs to store the `c_custkey` column, each materialized tuple has eight bytes.

Duplicate Operators. Often, the same operator type appears multiple times within a single pipeline. Most commonly, pipelines contain multiple successive join probes. Note that only probe or pass-through operator stages can be used repeatedly. Most operators with very complex nonlinear cost are materializing operators and appear only once per pipeline (sort, group-by, join-build, ...). To fit the features of duplicate operator stages into a fixed sized feature vector, we use feature addition. To enable this, we carefully designed our basic features to maintain meaning after being summed up. To indicate duplicate operators, we also add a *count* feature for each operator. If a stage type does not appear in the pipeline, the count feature will remain at 0. For each occurrence, we increment this feature value by one. We can see this in the features of TPC-H query 5 pipeline 2 in Listing 4. There are two hash join probe phases and the *HashJoin_Probe_count* feature is set to 2. Next, our percentage feature, as described above, indicates the likelihood of a single tuple of reaching a certain operator stage. The sum of several percentages forms the expected number of stages a tuple will reach. In the Q5 Pipeline 2, all tuples reach the first probe, but only 3% satisfy the join predicate and reach the second probe. The expected value of probes per tuple will be 100% + 3% = 103%, the value we compute for *HashJoin_Probe_right_percentage*. Although this summation of features loses some information, it is a meaningful input representation that works well in practice.

Table Scan Operators. Table scans are very important operators for performance prediction, as they commonly make up a large part of the execution time. We use input cardinality and output percentage features, as described above, to account for selections pushed down into the operator. Furthermore, we add additional features for the different kinds of filter predicates in table scans. We create a feature for simple comparison operations, like-expressions, between-expressions, and all other observed expression types. For each of those, we add the percentage of tuples for which they are evaluated. For example, in Listing 3 the table scan selects all Asian customer tuples. Umbra optimized the join with region and nation away and checks *c_nationkey* directly. Additionally, it optimized this check into two expressions. First, tuples are checked to be within 8 and 21 by a between-expression. Only if they qualify, they are checked against the actual values (8, 9, 12, 18, 21) with an in-expression. Hence, our feature vector shows that all tuples reach the between-expression with *TableScan_Scan_between_percentage* = 100%. Only 60% of the tuples reach the in-expression and only 20% of the tuples are selected and pushed to the next operator.

Little Manual Work. We aim to minimize the required effort to add new operators to T3. To do so, we define the features for each operator stage as a list of required basic features. T3 then automatically compiles all required features into a feature vector and assigns indices to each feature. This process allows us to add new operators with minimal manual effort.

4 Generating and Benchmarking Queries as Training Data

Machine Learning Requires Data. Machine learning models recognize patterns from their training data. If those patterns generalize well, they can yield good results for new data points. For our purpose, this means we need to train our model on a large dataset of benchmarked queries.

Training Data Distribution. Ideally, we could sample random queries from the distribution of all workloads that the model will be used on in the future [38]. We would need both, the data of database instances and the queries that are used. This is impossible because real-world workloads are not available to us. Hence, we randomly generate queries to create a training dataset that covers a wide spectrum of potential queries. We generate queries for a variety of database instances to enable the model to generalize across instances.

4.1 A Multitude of Database Instances

Data. We can maximize the coverage of the possible query space by using more database instances. Hence, we use 21 publicly available database instances that are compiled by Hilprecht and Binnig for

their zero shot model [16]. This contains synthetic benchmarks such as TPC-H and TPC-DS on different scale factors as well as some real-world data covering financial, health, and other data.

Queries. JOB, TPC-H, and TPC-DS are benchmarks with both data and queries [23]. We add the provided queries of these benchmarks to our dataset of queries. However, these are not enough to form a train set. The larger part of our dataset consists of randomly generated queries for all instances.

Database Statistics. We collect statistics over the data for query generation. First, we parse the SQL schema definition of an instance to obtain table names as well as column names and types. Then, we run some queries against the database to retrieve table cardinalities, distinct counts, and value ranges for numeric types. Finally, we find potentially join-able columns from different tables by considering their names and types. Our approach of using automatically retrieved features to generate queries is scalable and allows to add new instances with minimal effort.

4.2 Randomly Sampling Diverse Queries

Query Structure. As stated above, we aim to provide a query set that is as diverse as possible. Previous approaches generate random queries that adhere to a rather strict structure [16, 20]. In contrast, we build queries modularly from the following primitives:

- **Filter**: A table scan followed by some filter predicates. We use both, simple numeric filters to vary the percentage of qualifying tuples, as well as more complex predicates such as like patterns or between expressions and more.
- **Join**: A set of tables that are connected by possible join columns.
- **Aggregate**: A collection of aggregate functions that combine all tuples to a single row or group them by a suitable column.
- **Sort**: A sort operator that orders arbitrary input on suitable columns.
- **Project**: A random subset of available columns that are included in the output.

We can combine each of those primitives to create new query structures. In our query set, we explicitly generate differently structured groups of queries. For example, we have a group of queries that only include filters and projections. They all scan a single random table, use random filter predicates, and return a column subset. There is also a group combining all primitives. Each query of this group scans several tables and performs some filter predicates on them. It joins all of those on suitable columns and groups the output by some attributes. Finally, it sorts the resulting values by any column and selects a subset of the columns as output.

Dataset of Queries. For each of the 16 query structures, we generate 40 queries per database. Overall, we generate about 14,000 queries. To evaluate our model, we use all queries from the TPC-DS database instance to form a test set. As we have three variants of the TPC-DS database instance with scale factor 1, 10, and 100 we have over 2000 queries for evaluation. There are 13,000 queries we use for training. The train set does not include any TPC-DS queries.

4.3 Benchmarking

Reliability of Benchmarks. To ensure reliable execution time measurements we run each query 10 times and use the median running time. For additional information such as cardinalities, we also run an *explain analyze* statement. Finally, we store 10 running times and one analyzed plan for our experiments. Hence, we can also evaluate the deviations observed in our benchmarks. Table 3 shows statistics over the largest q-errors. Out of the 10 runs, we compare $\frac{2}{3}$ (i.e., 7) that are closest to the median. We consider these statistics as a theoretical optimum for any performance prediction model. So we expect any model predictions to be at least 5.8% off on average, which is the observed running time variation. Note that repeated runs are not necessary for good model performance (see Section 5.7). In the following section, we evaluate T3 with regard to accuracy and speed.

Table 3: Deviations of benchmarks as q-error. For each query we consider the most consistent $\frac{2}{3}$ of all measured values and report the one that is the furthest from the median. This table shows statistics over all queries. For 90% of all queries, the measured running time deviates by less than 13%.

p10	p50	p90	p95	p99	Avg
1.002	1.036	1.129	1.191	1.400	1.058

5 Evaluation

We ran all our benchmarks on an Ubuntu 24.04 machine using Linux 6.8.0-39-generic with 64 GiB of DRAM and a Ryzen 9 5950X processor with 32 threads on 16 cores locked at 2.8 GHz.

5.1 Prediction Speed

Fast. As depicted in Table 1 and Table 2 T3 is remarkably fast. The advantages in speed come from two main factors. First, we use decision-tree-like models that are inherently faster than neural networks. Second, we compile our model to native machine code using the LLVM-based compilation framework *lleaves* [3]. This native machine code executes much faster than regular decision tree frameworks.

Simple. Another benefit of a fast model is that one could replace complex hierarchical models like Stage with a single component [50]. T3's prediction latency is closer to a cache hit in Stage than to the latency of Stage's other models. Furthermore, it prevents the rare but very expensive calls to its slow global model, which would increase tail latencies. It also has the advantage that a single component is significantly easier to maintain than a whole model hierarchy.

Scalable. T3 uses one feature vector per pipeline in a query. Consequently, the prediction latency increases for queries with more pipelines. In our dataset, we observed an average of 4 and a maximum of 45 pipelines per query. Figure 5 shows the prediction latency of a single-threaded compiled implementation. For this benchmark, we randomly select 1 to 1000 pipelines, since many random pipelines perform equivalently to a large query for T3. We observe that the prediction time scales linearly from about 1.5us to 700us for 1000 pipelines. In contrast, the interpreted version is much slower when only using a single thread. Interpretation using multiple threads can be faster than the compiled single thread model for very large queries. However, we expect nearly all real-world queries to have fewer than 100 pipelines. For these, multi-threaded interpretation does not improve latency compared to the compiled

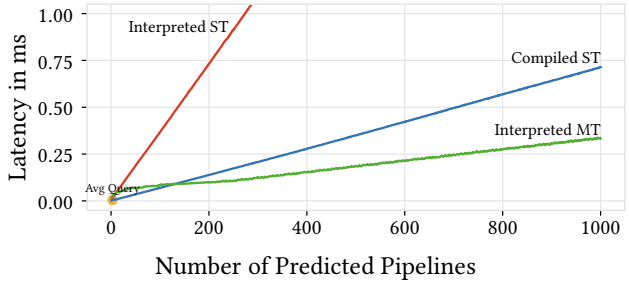

Figure 5: T3's prediction latency by number of pipelines. Compiled T3 is always single-threaded (ST), interpretation can also be executed multi-threaded (MT). The average query consists of 3 pipelines and has a prediction latency of 4us.

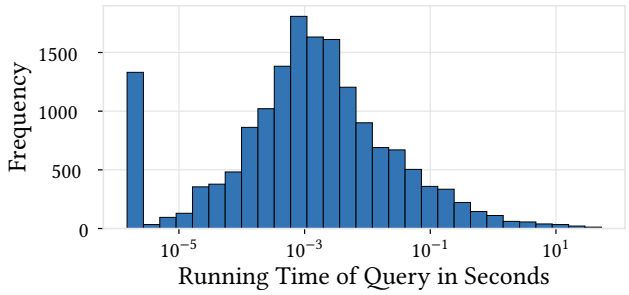

Figure 6: Observed running times of queries in our dataset.

model. Also, the compiled version could easily be parallelized as well, but single-threaded execution leaves the remaining CPU cores available for other workloads. Hence, we consider the compiled version to be the best pick. Note that neural network models that rely on the graph structure of query plans also suffer from increased execution times for larger queries. We conclude that T3 scales to extremely large queries while maintaining low prediction latencies. In the following sections, we will show that its accuracy is competitive to state-of-the-art approaches.

5.2 Dataset of Queries and Benchmarks

Running Times. As depicted in Figure 6, the running times of queries in our dataset vary drastically. The most common running times are around one millisecond. But the longest running queries run for over 20s while the shortest finish in less than 2us.

Very Short Running Queries. Figure 6 also shows a spike of many queries that run for a very short time. This usually occurs for two main reasons. First, some queries are extremely selective and produce very small or empty intermediate results. Second, the database's optimizer can sometimes answer the query without starting the execution engine. Optimizations like early-execution and predicate simplification can determine very small or empty intermediate results. In our experiments T3 did not show any notable accuracy anomalies for these queries.

Queries	p50	p90	Avg
Train Queries	1.14	1.69	1.32
All TPC-DS Test Queries	1.19	1.95	1.46
TPC-DS Benchmark Queries	1.30	2.77	1.94
TPC-DS sf 100 Test Queries	1.25	2.16	1.57
TPC-DS sf 100 Benchmark Queries	1.48	3.30	2.12

Table 4: Accuracy of T3 measured in q-error. T3 is trained on 20 training database instances and has never seen any information about TPC-DS data or queries.

Q-Error Percentiles and Average. We employ q-error as an evaluation metric for meaningful comparison [35]:

$$\text{q-error}(a, b) = \max\left(\frac{a}{b}, \frac{b}{a}\right) \tag{2}$$

Q-error penalizes overestimation and underestimation equally and is easy to interpret. Further, we aggregate the q-errors of many queries in three ways: Regular averages as well as p50 and p90 percentiles. As we will show in the following, there can be heavy outliers for performance prediction. Percentiles and averages allow us to reason about the whole accuracy distribution.

5.3 Accuracy on Test Set

Accuracy. First, we show the prediction accuracy of T3 for exact cardinalities in Table 4. Results with predicted cardinalities are in Section 5.6. T3 was trained on 20 database instances but has never seen any information about TPC-DS data or queries. First, we report the accuracy on the training queries. From the low average q-error of about 1.3, we conclude that the model can capture relevant information from the provided feature vectors. Second, we provide the accuracy for our whole test set of queries for TPC-DS database instances. These test queries contain over 1500 generated queries of all groups, as described in Section 4.2, as well as the 100 queries of the original TPC-DS benchmark. The test queries were run on three TPC-DS database instances with scale factors 1, 10, and 100 respectively. With an average q-error of about 1.5, the accuracy on this unseen workload and environment is only moderately worse. We consider this a good result and we show in Section 5.4 that it is competitive to state-of-the-art models. Clearly, T3 generalizes to new data. Next, we show the accuracy for the 100 queries in the TPC-DS benchmark. It is slightly worse than the accuracy for our whole train set. Hence, we conclude that our generated query set is diverse enough to represent a wide range of workloads but doesn't fully cover the variety of the TPC-DS benchmark. Finally, we show the error only on the dataset with scale factor 100. We can see that predictions are slightly less accurate for these larger datasets.

Error Distribution. Figure 7 depicts the distribution of different prediction errors for our test set. We can see that the majority of queries are predicted with a small q-error. Furthermore, there are few but heavy outliers. This explains why the average q-error far exceeds the p50 accuracy in Table 4.

Query Types. Figure 8 shows the errors for different query types. We can see that there are some differences among query types. Especially, queries with selections, joins, and aggregations (SeJSia, CSeJia) can be predicted well. Predicting the queries that come from

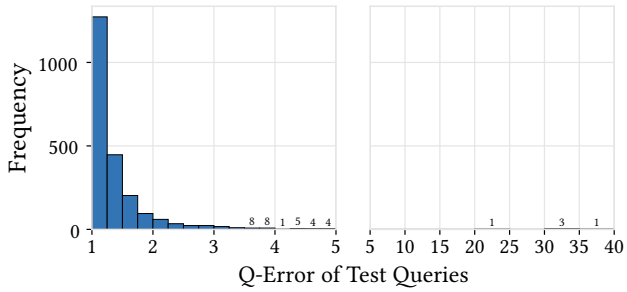

Figure 7: Frequency distribution of different q-errors for T3 predictions on all TPC-DS test queries.

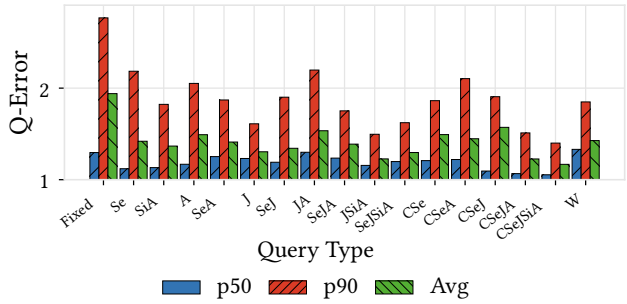

Figure 8: Different q-errors for different query types on TPC-DS data. Benchmark queries (Fixed) as well as generated queries: Selections (Se), aggregations (A), simple aggregations (SiA), joins (J), complex selections (CSe), window functions (W), and combinations of them.

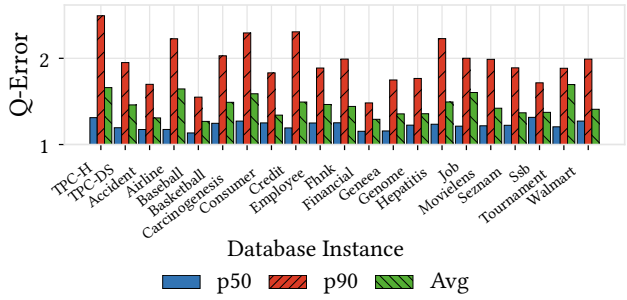

Figure 9: Different q-errors for different evaluation database instances. T3 is always trained on all but one database instance and then evaluated on the left-out instance. For TPC-H and TPC-DS, the respective numbers include different scale factors.

the original TPC-DS benchmark (Fixed) are hardest to predict.

Database Types. Figure 9 shows the q-errors for different database instances. Again, T3 is never trained on the evaluation instances on. The median (p50) q-error does not vary much across databases. Hence, T3 shows robust generalization across all database instances we used. The p90 and average q-errors vary a bit more.

5.4 Accuracy Comparison

Compared Models. In this section we compare T3's accuracy to state-of-the-art models. We provide a realistic comparison to Zero Shot [16] and explain why a comparison to AutoWLM and Stage is not meaningful. Here an overview of the models we show in Figure 1:

(1) **T3 (ours)**, a compiled decision tree that is trained on randomly generated queries on 20 database instances running on Umbra. Evaluated on all TPC-DS test queries.
(2) **Zero Shot** cost models as proposed by Hilprecht and Binnig [15, 16]. A neural network that is trained for the PostgreSQL database using a huge set of randomly generated queries for almost the same database instances as T3. Evaluated on JOB-full queries that have a very similar pattern to the generated training queries.
(3) **Stage**, a hierarchical model for Amazon Redshift [50]. It combines caches, decision trees, and neural networks. The training and evaluation datasets for AutoWLM and Stage are traces from the workload observed at Redshift over three weeks across 100 instances.
(4) **AutoWLM**, the predecessor method of Stage [40]. A workload-based decision tree model that was replaced to increase accuracy and handle changing workloads.

Comparison Problems. All of these approaches are created for different database systems. This means that various factors can affect the performance prediction problem. For example, parallel and distributed execution strategies, query compilation or interpretation, storage medium access, and database operator implementations all affect the performance behavior of the systems. It is not clear that predicting performance for different systems is equally hard. Furthermore, all systems are trained and evaluated on different queries. As shown in Figure 8, the kinds of queries can heavily influence the prediction accuracy. For example, Zero Shot does not work for complex workloads like TPC-DS that are also harder to predict, so its accuracy appears better. Finally, for comparability, we use estimated cardinalities for all systems. Consequently, inaccurate cardinality estimates can further impact the predictions.
Neither Stage nor AutoWLM are publicly available. Therefore, we cannot reproduce results for these models. Also, the workload used for their evaluation is unavailable. So every try to provide a comparison would depend on the workload we choose to evaluate T3. As the workload has a tremendous impact on the accuracy, we decide not to provide a direct comparison.

Accuracy Comparison to Zero Shot. As Zero Shot model implementations are available, we can reproduce the results for comparison with T3 in a reasonably comparable way. To eliminate the effects of cardinality estimation, we use perfect cardinalities. Furthermore, we evaluate on the same queries: the Join Order Benchmark [23], also called JOB-full. The remaining difference is that both models predict the performance of different database systems. JOB-full contains queries with several joins and complex selection predicates. We trained Zero Shot using their *complex workload* as training queries, which follow a very similar pattern that consists of selective table scans, equi-joins, and a final aggregation to a single tuple. Figure 10 shows that T3's accuracy actually matches Zero

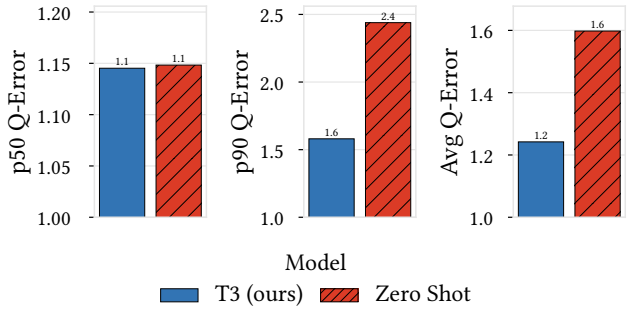

Figure 10: Accuracy comparison of Zero Shot model and T3. Both are trained on other database instances and evaluated on the queries of the Join Order Benchmark.

Table 5: Optimization time of join ordering with DPsize using T3 compared to a very simple cost function. Optimizes all 113 queries of the Join Order Benchmark.

Cost Model	Opt. Time	Model Calls	Time/Call
C_{out}	8.5ms	158320	0.054us
T3	525.4ms	316640	1.659us

Shot models in this case. The median (p50) q-error of T3 is approximately equal, p90 and Avg q-errors are better. We also conducted the same comparison with estimated instead of exact cardinalities. T3's accuracy is superior in all metrics in that experiment. We conclude that neural network models yield excellent accuracy. Yet, T3 matches and even surpasses this accuracy in comparable settings.

5.5 Join Ordering Microbenchmark

Join Ordering Implementation. To validate the applicability of T3 to low latency use-cases, we implement a join order algorithm that uses T3 as a cost model. In particular, we implement DPsize [34] in C++ and apply it to the queries of the join order benchmark [23]. As a baseline, we use DPsize with C_{out} as introduced by Cluet and Moerkotte [10]:

$$C_{out}(\mathcal{T}) = \begin{cases} 0 & \text{if } \mathcal{T} \text{ is a leaf} \\ |\mathcal{T}| + C_{out}(\mathcal{T}_1) + C_{out}(\mathcal{T}_2) & \text{if } \mathcal{T} = \mathcal{T}_1 \bowtie \mathcal{T}_2 \end{cases} \quad (3)$$

This is a very simple cost function that can be computed by adding three values in DP algorithms. We use a cardinality oracle to provide correct cardinalities with low latency for both variants. Hence, our implementations should have minimal overhead and should stress the performance impact of calling T3.

Optimization Speed. Optimizing all 113 queries using T3 is about 60x slower in our implementation. Exact numbers are listed in Table 5. On average, a call to T3 takes less than 2us. This is faster than the expected average we provided in Table 1 because each call predicts only one pipeline. Furthermore, there are twice as many calls to T3 than to C_{out}. This is because for every new subtree, two pipelines change. We add the build stage of a hash join to the open pipeline of the left subtree and a probe stage to the open pipeline of the right subtree. We cache the cost for all other pipelines that already finished in the subtrees. Overall, we can see that using T3 for algorithms that call it very often is feasible. Nonetheless, it is still comparatively slow for join ordering.

Table 6: Execution time of all JOB queries. Join orderings generated by our DPsize implementation using C_{out} and T3. For reference we also include Umbra's native optimizer.

Cost Model	Execution Time
C_{out}	1.348s
T3	1.366s
Native DB	1.382s

Resulting Trees. In this section, we validate that our join ordering algorithm actually works with T3. To do so, we force Umbra to execute the queries with the join ordering found by our algorithm. Even though C_{out} is very simple, we expect it to work very well for this use-case. Minimizing the sizes of intermediate results should be a near-perfect strategy. Note that we use the correct cardinalities for both C_{out} and T3. While Umbra has to use the join order provided by the plan, it still performs simple optimizations that do not change the structure. For example, it chooses to build hash tables over the smaller of the two inputs and probe the larger side. Hence, the symmetric structure of C_{out} is no disadvantage. We therefore consider equal execution times for both models a good result. As shown in Table 6, the plans found using T3 are slightly worse than using C_{out}. The total execution time of all[2] JOB queries with T3's plans is about 1.6% longer than C_{out}'s plans. In most cases both algorithms result in the exact same plans. For eight queries, T3 provides slightly faster plans, while for 16 queries C_{out}'s plans are slightly faster. We observed that T3 sometimes prefers plans with lower depth but larger intermediate result size, as long as these intermediate results are not materialized. Umbra's default join order algorithm yields slightly slower plans. The reason for this is that it does not have access to the true cardinalities during optimization and has to rely on statistics for cardinality estimation. This experiment validates that T3 gives reasonable estimates. It also shows that C_{out}, although unusable to predict execution time, is still a good metric for join ordering.

Performance Prediction Not Useful For Join Ordering. We argue that join ordering is not a very compelling use-case for elaborate cost models like T3. Leis et al. state that very simple cost models are suitable for yielding good join orderings [23]. The reason for this is that join ordering algorithms do not need to know how long individual plans will take to execute. Instead, they only need a reliable metric to compare two plans to each other. Minimizing intermediate result sizes is intuitively a sensible strategy. Furthermore, details in the cost model will usually be outweighed by cardinality estimation errors. The potential for improvement with different cost models is thus limited. Additionally, join ordering algorithms tend to call the cost model tremendously often. T3 is fast enough to be used for optimizing moderately large queries. However, we believe that adaptive optimization frameworks will benefit more from covering larger search spaces with simple cost

[2]Note that we excluded the results of query 33 because our forced query plans do not correctly distinguish between self-joining relations and the results are not reliable.

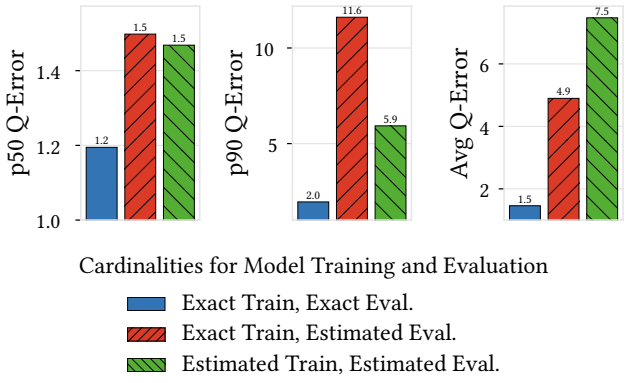

Figure 11: Accuracy with perfect and estimated cardinalities. Three variants: 1. A model trained on perfect cardinalities and evaluated on perfect cardinalities. 2. Same model trained on perfect cardinalities and evaluated on estimated cardinalities. 3. Model trained on estimated cardinalities and evaluated on estimated cardinalities. Median (p50) q-error left, p90 q-error middle, and avg q-error right. Evaluation on all TPC-DS test queries on scale factors 1, 10, and 100.

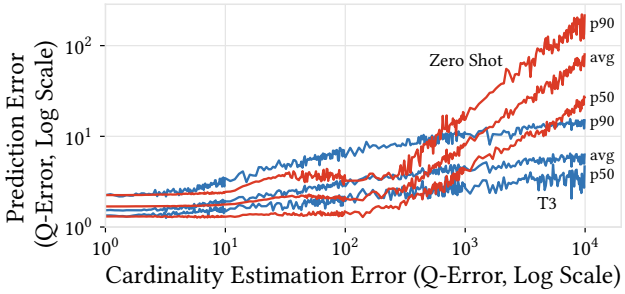

Figure 12: Accuracy of T3 and Zero Shot for artificially degraded cardinality estimates. Going from exact cardinalities to 1000x distorted cardinalities.

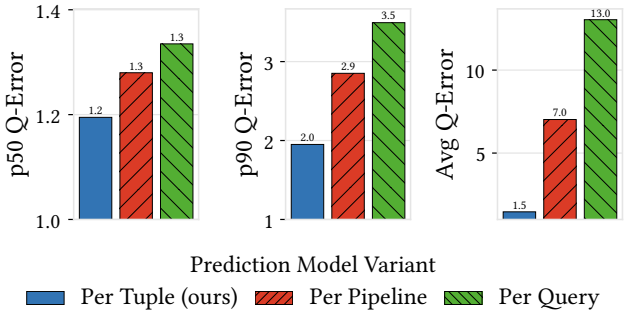

Figure 13: T3 accuracy when predicting for a single tuple, an individual pipeline, or a whole query. The second variant tries to directly predict the execution time for the whole pipeline. The third variant predicts on a single feature vector that is the sum of all pipeline feature vectors.

models than from using more complex cost models and introducing greedy algorithms earlier [37]. Hence, we consider T3 useful for other low latency use-cases as listed in Section 1, but do not recommend it as a join ordering cost model.

5.6 Cardinality Estimates

Predicted Cardinalities. As described in Section 2.1, we use perfect cardinalities to train and evaluate T3. We argue that cardinality estimation is an orthogonal problem that is out of scope for this work. A lot of research is done on better cardinality estimation methods. For example, Hilprecht and Binnig used DeepDB cardinality estimations to improve their model [16, 18]. If those methods get better, T3 will benefit directly as well. Nevertheless, in real use-cases we do not have correct cardinalities at hand. So we provide accuracy numbers for estimated cardinalities in Figure 11. First, we see that the median q-error degrades for imperfect cardinality estimates as expected. Second, the p90 q-error increases significantly. Large cardinality estimation errors also result in large performance prediction errors. These large errors also strongly affect the average q-error. While the median q-error only changes slightly, the average multiplies. This was also observed for Stage [50]. Furthermore, we see that for most queries T3 trained on estimated instead of perfect cardinalities is more accurate. T3 successfully compensates for cardinality estimation errors in many cases. However, the larger average error indicates that there a few very heavy outliers, which do not exist when trained on exact cardinalities. We conclude that while the model can compensate for wrong cardinality estimates in many cases, it fails worse when it cannot. We observe that incorrect cardinality estimates distort performance prediction accuracy substantially. Therefore, these numbers are not very suitable for a good judgement of T3's prediction quality. Once again, better cardinality estimation remains an important and unsolved problem. Furthermore, we argue that future work on better cardinality estimates is the most promising direction for improvements in performance prediction.

Accuracy Under Degrading Cardinality Estimates. To investigate T3's resilience against cardinality estimation errors, we show the degradation of model accuracy under increasingly distorted cardinality estimates. To do so, we manually modified the cardinalities by increasing factors. Figure 12 shows that with increasing errors in cardinality estimates the accuracy of both T3 and Zero Shot decreases drastically. While both models start at roughly the same accuracy, T3's prediction error increases slightly faster for small errors in cardinalities. However, when cardinalities are very imprecise, roughly 500x, Zero Shot starts to degrade worse than T3.

5.7 Ablation Study

In this section, we investigate how individual aspects of T3 contribute to its excellent prediction accuracy. In all following experiments, T3 is trained on all except the TPC-DS database instances and evaluated on all TPC-DS test queries with exact cardinalities.

Per Pipeline Feature Vectors. One of the most novel ideas of

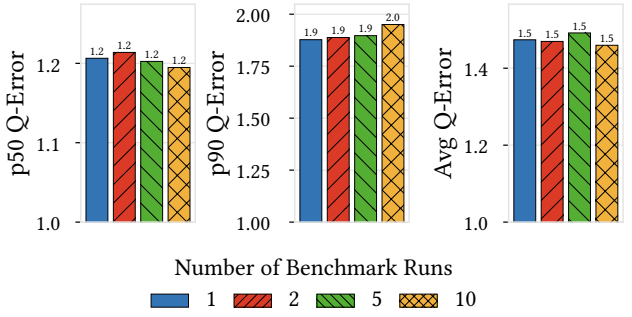

Figure 14: Model accuracy for different number of benchmark runs. There is no evidence that multiple benchmark runs improve model performance significantly.

T3 is to predict performance for each pipeline. Our decision tree model can only work with flat feature vectors. By decomposing queries into several feature vectors for each pipeline, we still enable indicative feature vectors for the workload. In this experiment, we create a variant of T3 that uses only a single feature vector per query. We simply use the sum of all per pipeline feature vectors. Figure 13 shows that T3 is much more accurate when predicting for individual pipelines. As described in Section 5.1, multiple predictions for single queries result in higher prediction latency. However, we argue that prediction latency is good enough for most use-cases and that the gain in accuracy is worth the overhead of multiple predictions.

Per Tuple Prediction. As described in Section 2.4, T3 predicts the running time for a single tuple in a pipeline. Note that this is only possible when we predict the execution time for each individual pipeline. Traditionally, similar methods try to estimate the running time for the whole pipeline directly. Due to the architecture of decision trees, it is hard for them to precisely predict values over a large range. Predicting per-tuple times instead automatically scales the prediction to different relation sizes. Figure 13 validates our intuitive explanation. T3 works considerably better when predicting execution times for single tuples.

Repeated Benchmarks. Finally, we investigate the effect of clean benchmark data for training. We trained T3 with benchmarks that are each executed 10 times. Then, the median execution time is used as training targets. Of course, this increases the required time to run benchmarks. This is not an issue for a research project like this work, as we need to collect training data only once. However, it might be useful to shorten the benchmark time for some use-cases, such as creating models for a variety of hardware. As depicted in Figure 14, there is no clear advantage to running several benchmarks per query. The pure running time of all queries running once hence decreases to just over 6 minutes. With the additional overhead of running queries and requesting analyzed plans, we estimate that the benchmarking time could be reduced to about 15 minutes.

6 Limitations

Although we consider T3 a very practical performance prediction model, our implementation has some limitations.

Spilling to Disk Not Considered. Umbra does not implement operators that spill intermediate data to disk. Consequently, our implementation of T3 does not cover the additional cost of spilling operators. Nevertheless, T3 should implicitly capture the effects of spilling analogously to the cost of materializing data in memory. Further, T3's architecture allows one to add new features to model spilling better.

Resource Congestion Not Considered. In our experiments, we do not run queries concurrently. Hence, T3 will always predict the expected execution time of the query when no concurrent queries are running. We think that it would be possible to model concurrency using new features and training data to some degree. However, we consider workload-centric approaches like the work of Zhou et al. [56] more promising for this goal.

Resource Usage Modeled Implicitly. We do not use explicit resource usage features like disk reads vs. cached data. T3 models disk reads implicitly, as it observed scans over large data to take longer in the training data. Table 4 shows that although predictions for larger datasets are slightly less accurate, T3 can still predict for queries that scan larger data.

Hardware Specific Model. T3 is trained on the observed execution time of benchmarked queries. To obtain these execution times, T3 must be trained for a specific machine to account for hardware specific characteristics. Adapting to new hardware requires running the queries on the new hardware. Since the number of different hardware setups is expected to be limited, even in cloud environments, we consider hardware-specific models to be the most practical solution. Running all queries we used for T3 on a commodity machine takes a few hours. In contrast, other approaches report much longer running times for benchmarks, e.g., 10 days [16] or three week traces of 100 instances [50].

Reliance on Cardinality Estimates. T3 uses cardinality estimates as prediction features. In line with the garbage in garbage out principle, T3's predictions degrade with poor cardinality estimates. This is a limitation it shares with other performance prediction models as depicted in Figure 12.

7 Related Work

7.1 Work on Performance Prediction

Main Memory System Cost Models. There has been considerable work on performance prediction over the past 20 years. Listgarten et al. identify main memory databases as a disruptive new technology and provide a first cost model [25]. Similarly, Manegold et al. identify hierarchical memory systems as key factors for performance prediction [28]. They present a generic approach to build cost models for such hierarchies.

Early Learning-Based Methods. Before the widespread adoption of neural networks, there were many attempts on performance prediction using other learning-based methods. Gupta et al. use a technique very similar to decision trees to predict time ranges for queries as early as 2008 [13]. Akdere et al. use support vector machines and linear regression models on both operator-based and query-plan-based features [1]. Wu et al. calibrate PostgreSQL's cost model using the observations of a small set of calibration queries [48]. In subsequent work they use an analytical model to combine the results of the calibrated cost model for multiple queries

and predict the execution time of whole workloads [47]. They show that their approach is very competitive to full machine-learning-based approaches of the time, especially for dynamic workloads of unpredictable queries. Later, they add uncertainty measures for cost prediction by modeling constants as random variables [49].

Workload-Based Neural Network Models. When neural networks became more common, they were applied as estimators for fixed workloads. Marcus and Papaemmanouil propose to use deep learning for performance prediction [32]. They hand-craft a neural network architecture that can pass *interesting* data between operator-level neural units. Sun and Li compare different methods of applying deep learning to the tree shaped query plans [43]. They observe a pooling architecture to be effective for combining nodes of trees. Zhao et al. improve on this using a transformer architecture and available database statistics to the model input [55]. Zhou et al. employ a graph embedding to encode both single queries, but also multiple concurrently running queries [56]. By modeling the interactions of concurrent queries, they accurately predict the execution time of whole workloads. Chang et al. compare a variety of deep learning architectures and introduce a combination of graph neural networks and gated recurrent units [5].

Modern Zero Shot Models. As workload based models come with several drawbacks, there has been work on models that can work on any database instance without new training. Siddiqui et al. build a cost model for a production workload to improve the optimizer of a big data system [41]. As this workload contains various very different queries from multiple customers, it is comparable to zero shot models. They use ensembles of many different models for different parts of queries and combine them with a gradient boosted decision tree. Hilprecht and Binnig pioneer zero shot models by purposefully training and evaluating them on different instances [15, 16]. Despite this challenging setup, they achieve excellent accuracy. Heinrich et al. apply a similar model to distributed stream processing [14]. Wu et al. build Stage, a model for the entire Redshift workload, which arguably needs to generalize to all kinds of new database instances [50]. They leverage the accuracy of neural networks but try to improve the model latency using a model hierarchy.

Different Approaches. In this paragraph, we display work that significantly deviates from the learning-based approaches mentioned above to solve the performance prediction problem. Before their work on zero shot models, Hilprecht et al. propose cost models based on differential programming [17]. They re-use the known functions of cost models and only learn a few constants within those functions similar to [48]. Boissier uses linear regression models to predict the performance of different compression schemes for dynamic encoding selection [4]. Yang et al. provide multiple sets of calibration constants for fixed cost functions based on the PostgreSQL cost model [51]. They select the best constant set on a query basis with a decision tree model. As none of the above methods rely on neural networks, they are comparatively quick to train and evaluate.

7.2 Other Learning Methods for Databases

Generating Workloads. There is extensive work on randomly generating queries and workloads for learning methods on databases. Kipf et al. generate small join queries as training data for cardinality estimation models [20, 21]. Hilprecht and Binnig extend this method for more complex selection predicates [16]. Ventura et al. propose DataFarm to generate diverse workloads based on abstract execution plan patterns [46].

Prediction of Other Metrics. There has been a plethora of work using learning-based cardinality estimation [20, 21, 24, 33, 42]. Furthermore, there is some work on estimating the progress of currently running queries [6, 7, 26]. Finally, there is much work on learned query optimizers [2, 29, 30, 52, 57].

Latency Problems. Latency of learning-based prediction methods has been identified as an issue repeatedly in previous work [22, 50, 51]. Lehmann et al. conclude that many modern query optimization techniques that rely on learned models fail to improve end-to-end execution time due to prolonged optimization times [22]. Wu et al. hide the latency of neural network models by using a query cache and a local decision tree model instead where possible [50]. Many other models that do not rely on slow machine learning models provide comparatively good latencies [4, 17, 47, 48].

8 Conclusion

In this work, we propose T3, an extremely fast cross-database performance predictor with state-of-the-art accuracy. We show how to achieve good prediction accuracy with pipeline-based query plan representations and tuple-centric prediction targets. Further, we demonstrate the use of decision tree compilation to native machine code for better latencies. T3 makes the application of accurate performance prediction models practical for latency-sensitive use-cases. We conclude that current models tackle the problem of performance prediction very well, given good cardinality estimates. Further work on cardinality estimation has the greatest potential to improve performance prediction.

References

[1] 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, 390–401.

[2] Christoph Anneser, Nesime Tatbul, David E. Cohen, Zhenggang Xu, Prithviraj Pandian, Nikolay Laptev, and Ryan Marcus. 2023. AutoSteer: Learned Query Optimization for Any SQL Database. *Proc. VLDB Endow.* 16, 12 (2023), 3515–3527.

[3] Simon Boehm. 2021. *lleaves*. https://github.com/siboehm/lleaves

[4] Martin Boissier. 2021. Robust and Budget-Constrained Encoding Configurations for In-Memory Database Systems. *Proc. VLDB Endow.* 15, 4 (2021), 780–793.

[5] Baoming Chang, Amin Kamali, and Verena Kantere. 2024. A Novel Technique for Query Plan Representation Based on Graph Neural Nets. In *DaWaK (Lecture Notes in Computer Science, Vol. 14912)*. Springer, 299–314.

[6] Surajit Chaudhuri, Raghav Kaushik, and Ravishankar Ramamurthy. 2005. When Can We Trust Progress Estimators for SQL Queries?. In *SIGMOD Conference*. ACM, 575–586.

[7] Surajit Chaudhuri, Vivek Narasayya, and Ravishankar Ramamurthy. 2004. Estimating progress of execution for SQL queries. In *SIGMOD Conference*. ACM, 803–814.

[8] Jing Chen, Tiantian Du, and Gongyi Xiao. 2021. A multi-objective optimization for resource allocation of emergent demands in cloud computing. *J. Cloud Comput.* 10, 1 (2021), 20.

[9] Yun Chi, Hyun Jin Moon, Hakan Hacigümüs, and Jun'ichi Tatemura. 2011. SLA-tree: a framework for efficiently supporting SLA-based decisions in cloud computing. In *EDBT*. ACM, 129–140.

[10] Sophie Cluet and Guido Moerkotte. 1995. On the Complexity of Generating Optimal Left-Deep Processing Trees with Cross Products. In *ICDT*, Vol. 893. Springer, 54–67.

[11] Archana Ganapathi, Harumi A. Kuno, Umeshwar Dayal, Janet L. Wiener, Armando Fox, Michael I. Jordan, and David A. Patterson. 2009. Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning. In *ICDE*. IEEE Computer Society, 592–603.

[12] Chetan Gupta, Abhay Mehta, and Umeshwar Dayal. 2008. PQR: Predicting Query Execution Times for Autonomous Workload Management. In *ICAC*. IEEE Computer Society, 13–22.

[13] Chetan Gupta, Abhay Mehta, and Umeshwar Dayal. 2008. PQR: Predicting Query Execution Times for Autonomous Workload Management. In *ICAC*. IEEE Computer Society, 13–22.

[14] Roman Heinrich, Manisha Luthra, Harald Kornmayer, and Carsten Binnig. 2022. Zero-shot cost models for distributed stream processing. In *DEBS*. ACM, 85–90.

[15] Benjamin Hilprecht and Carsten Binnig. 2022. One Model to Rule them All: Towards Zero-Shot Learning for Databases. In *CIDR*. www.cidrdb.org.

[16] Benjamin Hilprecht and Carsten Binnig. 2022. Zero-Shot Cost Models for Out-of-the-box Learned Cost Prediction. *Proc. VLDB Endow.* 15, 11 (2022), 2361–2374.

[17] Benjamin Hilprecht, Carsten Binnig, Tiemo Bang, Muhammad El-Hindi, Benjamin Hättasch, Aditya Khanna, Robin Rehrmann, Uwe Röhm, Andreas Schmidt, Lasse Thostrup, and Tobias Ziegler. 2020. DBMS Fitting: Why should we learn what we already know?. In *CIDR*. www.cidrdb.org.

[18] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, not from Queries! *Proc. VLDB Endow.* 13, 7 (2020), 992–1005.

[19] 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*. 3146–3154.

[20] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *CIDR*. www.cidrdb.org.

[21] Andreas Kipf, Dimitri Vorona, Jonas Müller, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2019. Estimating Cardinalities with Deep Sketches. In *SIGMOD Conference*. ACM, 1937–1940.

[22] Claude Lehmann, Pavel Sulimov, and Kurt Stockinger. 2024. Is Your Learned Query Optimizer Behaving As You Expect? A Machine Learning Perspective. *Proc. VLDB Endow.* 17, 7 (2024), 1565–1577.

[23] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *VLDB J.* 27, 5 (2018), 643–668.

[24] Pengfei Li, Wenqing Wei, Rong Zhu, Bolin Ding, Jingren Zhou, and Hua Lu. 2023. ALECE: An Attention-based Learned Cardinality Estimator for SPJ Queries on Dynamic Workloads. *Proc. VLDB Endow.* 17, 2 (2023), 197–210.

[25] Sherry Listgarten and Marie-Anne Neimat. 1997. Cost Model Development for a Main Memory Database System. In *Real-Time Database Systems: Issues and Applications*, Azer Bestavros, Kwei-Jay Lin, and Sang Hyuk Son (Eds.). Springer US, Boston, MA, 139–162.

[26] Gang Luo, Jeffrey F. Naughton, Curt J. Ellmann, and Michael Watzke. 2005. Increasing the Accuracy and Coverage of SQL Progress Indicators. In *ICDE*. IEEE Computer Society, 853–864.

[27] Chenghao Lyu, Qi Fan, Fei Song, Arnab Sinha, Yanlei Diao, Wei Chen, Li Ma, Yihui Feng, Yaliang Li, Kai Zeng, and Jingren Zhou. 2022. Fine-Grained Modeling and Optimization for Intelligent Resource Management in Big Data Processing. *Proc. VLDB Endow.* 15, 11 (2022), 3098–3111.

[28] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. 2002. Generic Database Cost Models for Hierarchical Memory Systems. In *VLDB*. Morgan Kaufmann, 191–202.

[29] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2022. Bao: Making Learned Query Optimization Practical. *SIGMOD Rec.* 51, 1 (2022), 6–13.

[30] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.* 12, 11 (2019), 1705–1718.

[31] Ryan Marcus and Olga Papaemmanouil. 2016. WiSeDB: A Learning-based Workload Management Advisor for Cloud Databases. *Proc. VLDB Endow.* 9, 10 (2016), 780–791.

[32] Ryan Marcus and Olga Papaemmanouil. 2019. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *Proc. VLDB Endow.* 12, 11 (2019), 1733–1746.

[33] Volker Markl, Guy M. Lohman, and Vijayshankar Raman. 2003. LEO: An autonomic query optimizer for DB2. *IBM Syst. J.* 42, 1 (2003), 98–106.

[34] Guido Moerkotte and Thomas Neumann. 2006. Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products. In *VLDB*. ACM, 930–941.

[35] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors. *Proc. VLDB Endow.* 2, 1 (2009), 982–993.

[36] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*. www.cidrdb.org.

[37] Thomas Neumann and Bernhard Radke. 2018. Adaptive Optimization of Very Large Join Queries. In *SIGMOD Conference*. ACM, 677–692.

[38] Andrew Ng. 2018. Machine learning yearning: Technical strategy for AI engineers, in the era of deep learning. (2018).

[39] Jignesh M. Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. 2018. Quickstep: A Data Platform Based on the Scaling-Up Approach. *Proc. VLDB Endow.* 11, 6 (2018), 663–676.

[40] Gaurav Saxena, Mohammad Rahman, Naresh Chainani, Chunbin Lin, George Caragea, Fahim Chowdhury, Ryan Marcus, Tim Kraska, Ippokratis Pandis, and Balakrishnan (Murali) Narayanaswamy. 2023. Auto-WLM: Machine Learning Enhanced Workload Management in Amazon Redshift. In *SIGMOD Conference Companion*. ACM, 225–237.

[41] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. 2020. Cost Models for Big Data Query Processing: Learning, Retrofitting, and Our Findings. In *SIGMOD Conference*. ACM, 99–113.

[42] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO - DB2's LEarning Optimizer. In *VLDB*. Morgan Kaufmann, 19–28.

[43] Ji Sun and Guoliang Li. 2019. An End-to-End Learning-based Cost Estimator. *Proc. VLDB Endow.* 13, 3 (2019), 307–319.

[44] Rebecca Taft, Willis Lang, Jennie Duggan, Aaron J. Elmore, Michael Stonebraker, and David J. DeWitt. 2016. STeP: Scalable Tenant Placement for Managing Database-as-a-Service Deployments. In *SoCC*. ACM, 388–400.

[45] Sean Tozer, Tim Brecht, and Ashraf Aboulnaga. 2010. Q-Cop: Avoiding bad query mixes to minimize client timeouts under heavy loads. In *ICDE*. IEEE Computer Society, 397–408.

[46] Francesco Ventura, Zoi Kaoudi, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2021. Expand your Training Limits! Generating Training Data for ML-based Data Management. In *SIGMOD Conference*. ACM, 1865–1878.

[47] Wentao Wu, Yun Chi, Hakan Hacigümüs, and Jeffrey F. Naughton. 2013. Towards Predicting Query Execution Time for Concurrent and Dynamic Database Workloads. *Proc. VLDB Endow.* 6, 10 (2013), 925–936.

[48] 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, 1081–1092.

[49] Wentao Wu, Xi Wu, Hakan Hacigümüs, and Jeffrey F. Naughton. 2014. Uncertainty Aware Query Execution Time Prediction. *Proc. VLDB Endow.* 7, 14 (2014), 1857–1868.

[50] Ziniu Wu, Ryan Marcus, Zhengchun Liu, Parimarjan Negi, Vikram Nathan, Pascal Pfeil, Gaurav Saxena, Mohammad Rahman, Balakrishnan Narayanaswamy, and Tim Kraska. 2024. Stage: Query Execution Time Prediction in Amazon Redshift. In *SIGMOD Conference Companion*. ACM, 280–294.

[51] Jiani Yang, Sai Wu, Dongxiang Zhang, Jian Dai, Feifei Li, and Gang Chen. 2023. Rethinking Learned Cost Models: Why Start from Scratch? *Proc. ACM Manag. Data* 1, 4 (2023), 255:1–255:27.

[52] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a Query Optimizer Without Expert Demonstrations. In *SIGMOD Conference*. ACM, 931–944.

[53] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: One Cardinality Estimator for All Tables. *Proc. VLDB Endow.* 14, 1 (2020), 61–73.

[54] Geoffrey X. Yu, Ziniu Wu, Ferdi Kossmann, Tianyu Li, Markos Markakis, Amadou Ngom, Samuel Madden, and Tim Kraska. 2024. Blueprinting the Cloud: Unifying and Automatically Optimizing Cloud Data Infrastructures with BRAD. *Proc. VLDB Endow.* 17, 11 (2024), 3629–3643.

[55] Yue Zhao, Gao Cong, Jiachen Shi, and Chunyan Miao. 2022. QueryFormer: A Tree Transformer Model for Query Plan Representation. *Proc. VLDB Endow.* 15, 8 (2022), 1658–1670.

[56] Xuanhe Zhou, Ji Sun, Guoliang Li, and Jianhua Feng. 2020. Query Performance Prediction for Concurrent Queries using Graph Embedding. *Proc. VLDB Endow.* 13, 9 (2020), 1416–1428.

[57] Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou. 2023. Lero: A Learning-to-Rank Query Optimizer. *Proc. VLDB Endow.* 16, 6 (2023), 1466–1479.

Received 17 October 2024; revised 23 January 2025; accepted 30 January 2025