# DBMS Fitting: Why should we learn what we already know?

Benjamin Hilprecht[1], Tiemo Bang[1], Muhammad El-Hindi[1], Benjamin Hättasch[1], Aditya Khanna[2], Robin Rehrmann[3], Uwe Röhm[4], Andreas Schmidt[5], Lasse Thostrup[1], Tobias Ziegler[1], Carsten Binnig[1]

[1] TU Darmstadt, Germany [2] IIT Bombay, India [3] TU Dresden, Germany [4] University of Sydney, Australia [5] KIT, Germany

## ABSTRACT

Deep Neural Networks (DNNs) have successfully been used to replace classical DBMS components such as indexes or query optimizers with learned counterparts. However, commercial vendors are still hesitating to put DNNs into their DBMS stack since these models not only lack explainability but also have other significant downsides such as the requirement for high amounts of training data resulting from the need to learn all behavior from data.

In this paper, we propose an alternative approach to learn DBMS components. Instead of relying on DNNs, we propose to leverage the idea of differentiable programming to fit DBMS components instead of learning their behavior from scratch. Differentiable programming is a recent shift in machine learning away from the direction taken by DNNs towards simpler models that take advantage of the problem structure. In a case study we analyze and discuss how to fit a model to estimate the cost of a query plan and present initial experimental results that show the potential of our approach.

## 1 INTRODUCTION

*Motivation.* Deep Neural Networks (DNNs) have not only shown to solve many complex problems such as image classification or machine translation, but are applied in many other domains, too. This is also the case for DBMSs, where DNNs have been successfully used not only for automatic database tuning [1, 20], but also to replace existing components with learned counterparts such as learned cost models [8, 18] as well as learned query optimizers [15, 16], learned indexes [5, 9], and learned scheduling or query processing schemes [14, 17].

The power of using DNNs results from the fact that DNNs represent heavily parameterized models that can approximate arbitrary functions. However, the black-box nature makes *DNNs hard to explain*; i.e., decisions of DNNs cannot really be inspected to understand how the learned algorithm is accomplishing its goals. For example, in the case of a learned cost model such as [18] that predicts the execution costs for a given query plan using black-box DNNs, a database administrator would not be able understand why the model produced a certain cost estimate. This is very different from classical cost models that estimate the costs of a plan by combining different factors such as cost of data accesses as well as processing costs. While these models are explainable and allow a database administrator to understand the decisions of the model they are hard to tune and often provide inaccurate estimates [11].


*CIDR'20, January 12-15, 2020, Amsterdam, Netherlands*

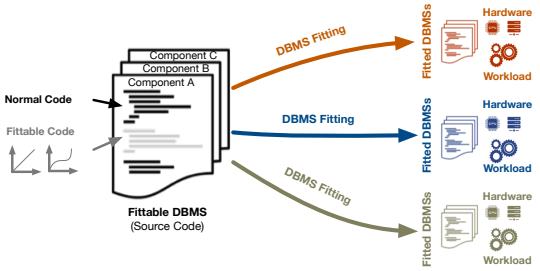



**Figure 1: A FITable DBMS - The idea is that the code base of a DBMS consists of fittable code that allows a DBMS to adjust its behavior to hardware and workload characteristics.**

Moreover, explainability is not the only reason why commercial vendors are hesitating to put DNNs into their DBMS stack[4]:

- First, *DNNs are data-hungry* since they have to learn even basic system behavior (that might be well known by a DBMS developer) purely from training data. For example, when learning a cost model, large training corpora are required which need significant time and resources to be constructed since each query in the training corpora needs to be executed and the execution time needs to be collected. Even worse, this is not a one-time effort, since the same procedure needs to be repeated for every new database that needs to be supported by the optimizer or if the current database is not static (i.e., the schema or data is changing).
- Second, it has been shown that *DNNs are susceptible to small changes* in the input; i.e., already a small change for one input feature can cause the DNN to produce erroneous predictions with high confidence. This is an effect that has also been shown by adversarial attacks. For DBMS, this problem cannot be ignored since it shows the general lack of DNNs to generalize to unseen input in a stable manner and to provide a robustness for DBMS components.
- Third, *DNNs are expensive to update.* In DBMSs, a learned component might need to be updated if the database or the workload changes. However, updating a learned component often requires collecting new training data and an expensive retraining of the DNN. While this might be acceptable in some cases (e.g., the retraining of a learned cost model might eventually be acceptable since it can be done offline), retraining might be too expensive for other components such as learned indexes that require online updates [9] if the database is dynamically changing as for OLTP workloads.

*Contributions.* In this paper, we propose a different route for learned DBMSs to tackle the aforementioned issues of black-box based approaches. Instead of relying on DNNs to replace classical

DBMS components, we propose to leverage the idea of differentiable programming[1] to implement *FITable DBMSs*. In a nutshell, differentiable programming is a recent shift in machine learning, away from the direction taken by DNNs that increasingly use heavily parameterized models and towards simpler white-box models that take more advantage of the problem structure [19]. Recently, differential programming has been used successfully to fit models in domains such as computer vision [12] to encode knowledge on how basic image processing primitives (e.g., edge detectors) work or to learn a physics engine [3] where differentiable functions encode the basic laws of physics that are then fitted.

The main idea of FITable DBMSs goes in the same direction where DBMS components (or parts) are implemented using differentiable functions as shown in Figure 1: Similar to normal code, differentiable functions implement logic inside a DBMS that already encodes the basic behavior of a component, but unlike normal code these functions in addition contain learnable parameters that allow to fit their behavior to a concrete workload and hardware. For example, query optimizers need to be able to estimate the physical cost (i.e., the total execution time) of query plans. Here, fittable functions could be used to describe the basic shape of cost functions for operators that could then be fitted to the behavior of the underlying hardware. While cost functions are a natural candidate for fitting, we believe that other components inside a DBMS such as data structures or execution strategies can benefit from fitting as we will discuss later.

FITable DBMSs are thus different from current approaches for learned DBMS components that purely rely on DNNs [10], since the behavior of a fittable component does not need to be learned from scratch. As a result, fittable functions not only need much less training data compared to training a DNN (which captures the same behavior) but also provide other benefits regarding the explainability and generalization as we show later in this paper. Another direction that is related to FITable DBMSs is the idea of synthesizing data structures [6] from known building blocks such as lists, dense arrays, zone maps, etc. to optimally support a given workload and hardware. Similar to fitting DBMS components, synthesizing data structures also generates explainable DBMS code. However, a major difference to fitting is that while synthesizing only targets the design of data structures, fitting is applicable to a broader set of DBMS components.

*Outline.* The remainder of the paper is organized as follows. In Section 2, we present our vision towards so called FITable DBMSs. Afterwards, in Section 3 we show by a concrete use case how the idea of fittable DBMS components could be used for cost estimation and present initial experimental results that show the benefits of our approach. Finally, we conclude in Section 4.

## 2 VISION: A FITABLE DBMS

### 2.1 Basic Idea of Fitting

The vision of a FITable DBMSs is that *DBMS components (or parts of them) are implemented as differentiable functions* that allow us to adapt the behavior of the component to optimally support a concrete workload and hardware. For instance, a simplified cost model

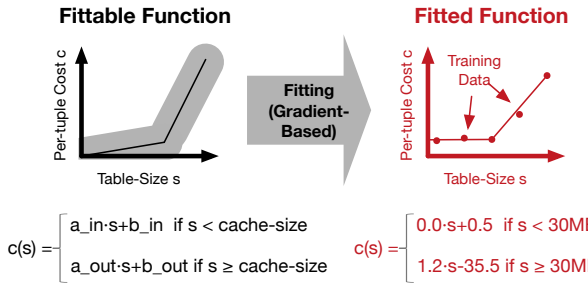



**Figure 2: Fitting a simple cost model for a scan operator to predict the per-tuple access cost - The example shows how the piecewise linear function can be fitted based on training data by learning the slope and intercept of each segment. For fitting we can use a gradient-based optimization method such as gradient descent.**

to estimate the execution time of a scan operator in a main-memory DBMS can be modelled as a differentiable function `cost_scan_op` as shown in Listing 1. The main idea of this function is that the costs for reading a tuple depend on the table size which can be represented by a piece-wise linear function using two segments for tables that fit into the cache and for those which spill out of the cache.

**Listing 1: Fittable Function for Simple Cost Model**

```
# table-size = size in Byte / no-tuples = number of tuple in table
def cost_scan_op(params, table_size, no_tuples):
    # piecewise linear model
    if table_size< params['cache-size']:
        slope = params['a_in']
        intercept = params['b_in']
        cost_per_tuple = slope * table_size + intercept
    else:
        slope = params['a_out']
        intercept = params['b_out']
        cost_per_tuple = slope * table_size + intercept

    return no_tuples * cost_per_tuple
```

The main benefit of fittable code is that it not only leverages the domain knowledge of the developer (e.g., that the tuple-access cost can be modelled as a piece-wise linear function in our example) but more importantly that the concrete behavior can be fitted automatically to the actual behavior. The fittable part of the code is captured by parameters that can be learned from concrete behavior. In our example, the learnable parameters are the slope (i.e., `params['a_in']` and `params['a_out']`) and intercept (i.e., `params['b_in']` and `params['b_out']`) of both segments.

For fitting the cost model, the actual costs of running the scan operator on different table sizes need to be collected. Since functions are differentiable, normal gradient-based optimization can be used to fit the parameters (i.e., minimize the error of the cost function) as shown in Figure 2. Once the parameters are fitted they can be used at runtime of a DBMS, just like fully specified source code.

The power of differentiable programming stems from the fact that the database developer does not have to come up with the gradients herself. Instead, frameworks such as Autograd[2] support

[1] https://www.facebook.com/yann.lecun/posts/10155003011462143

[2] https://github.com/HIPS/autograd

automatic differentiation [19] of ordinary code, which may contain all the usual control structures, including loops, if statements, recursion, and closures. In our example, the code for the cost function in Listing 1 is implemented using a normal if-else control flow that can be differentiated automatically.

Overall, fittable code in contrast to black-box DNN models thus provides many advantages: First, fittable code is more *data-efficient*, i.e. we require much less training data since the differentiable function already defines the basic shape of a function that needs to be learned. Furthermore, fitting a differentiable function does not always need to rely on gradient-based methods that typically require multiple passes over the training data. Instead, it can often be implemented by computationally much simpler approaches that only require a single pass [5]. Second, fittable functions typically *generalize* better and are less susceptible to small changes in the input, since they already define a reasonable behavior based on their shape. Finally, fitted code is *explainable and debuggable*. If the behavior is unexpected, the developer can debug the DBMS code (as usual) since the general code structure reflecting the domain knowledge is still interpretable and remains unchanged.

## 2.2 The Bigger Picture

There are many different directions the research community can investigate how fitting can be used to adapt DBMS components to a given hardware and workload. In the following, we discuss several DBMS components that are candidates for fitting but also propose directions regarding the general learning setup.

*Analytical Models.* In general, ideal candidates for fitting are DBMS components where (parametrizable) analytical models already exist. For example, transaction scheduling relies on models for conflict probability. While recent papers aim to learn the conflict probability using end-to-end ML models [17] there already exist analytical models [2] that define the conflict probability based on number of concurrent transactions, the database size, etc. However, these models typically rely on parameters that reflect the latency of lock requests as well as reads/writes. Fitting could be used to learn these parameters from the actual behavior of running these operations on a concrete hardware. Other ideas for which analytical models exist that can be fitted are software-prefetching or caching strategies that all rely on similar parameters to predict access costs.

*Data Structures and Algorithms.* While all the before-mentioned applications target the fitting of functions that model different notions of analytical models used in DBMSs for query optimization and scheduling (e.g., execution cost operations, conflict probability), we believe that fitting can be used also for other data structures and algorithms of a DBMS. For example, indexes such as B-trees can be seen as a function that predict the position of a key in a sorted array. While existing papers [9] use black-box DNNs to approximate this mapping function, in [5] we already showed that the idea of fittable white-box functions can be used to learn the mapping. Similar ideas for fitting that learn the data distribution can be used for learning algorithms of database operators such as sorting which is again in contrast to [10] which proposes to use black-box DNNs also for learning algorithms.

*End-to-end Learning.* Another interesting route that needs to be explored is how more complex models can be fitted end-to-end

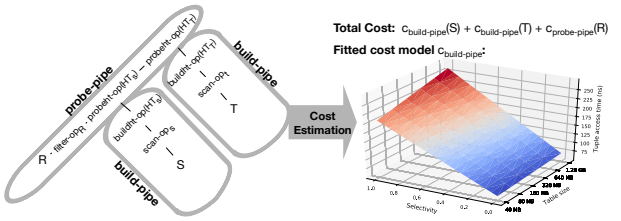



**Figure 3: Basic idea of our fittable cost model - The total cost of a query plan is estimated based on fitted cost models for each pipeline type. In this example, the build-pipeline type is used in two instantiations over tables $S$ and $T$ and the probe-pipeline type is used in one instantiation over table $R$, which probes into the hash tables $HT_S$ and $HT_T$, created by the other two pipelines. The cost models for each pipeline type are based on general features of a pipeline, such as the size of the input table, tuple-width, selectivity of operators etc.**

when using white-box fittable functions. End-to-end learning is typically seen as a major advantage of black-box DNNs which can combine several layers (e.g., fully-connected vs. convolutional) to capture complex behaviors. Differentiable programming makes the composition of complex models and end-to-end training also applicable for white-box models. The main idea is that similar to DNNS, white-box functions can also be combined into more complex models and auto-differentiation can then be applied to fit the composed models directly. For example, we will show later in this paper how a cost model to estimate the query execution cost of a complete plan is composed of cost models for individual operators that can be fitted end-to-end based on the monitored execution time of complete query plans.

*Grey-box Learning.* Finally, while fittable (i.e. white-box) functions provide many advantages over black-box DNNs, we still think that there is a need to combine both. The combination enables a DBMS to learn parts of components where the behavior can not easily be modeled as a fittable function or where the behavior is not known in advance. For example, it is hard to define fittable functions for cost models of operations that are allowed to call user-defined functions, since the complexity of the user-defined code can vary significantly. In this case, a normal DNN can be used to estimate the cost of the user-defined operation and still be combined with the fitted parts of the optimizer. Since fittable functions as well as DNNs are both differentiable, the composed model is still differentiable (due to the chain rule) and can be trained end-to-end.

# 3 CASE STUDY: A FITTABLE COST MODEL

In this section, we discuss the potentials of fitting by presenting a case study with a fittable cost model. In the following, we first discuss pitfalls of today's approaches for cost models, before we discuss how fitting can be applied to cost models to mitigate these issues. Afterwards, we show initial experimental results of our fitted cost model.

### 3.1 The Need for better Cost Models

Models that predict the execution cost of SQL queries are essential components of DBMSs. Query optimizers are the most well-known component that rely on cost models to choose between different alternative query plans based on cost estimations. However, this is not the only component in a DBMS that relies on cost models. More recently, papers have suggested to use cost models for self-driving databases [13] that automate physical design choices.

Traditionally, cost models are handcrafted in a DBMS and thus rely on detailed knowledge about the complexity of the underlying algorithm and data structures. However, these models are typically non-trivial to tune and often provide inaccurate estimates even when using automatic calibration tools [11]. And this is not the only obstacle of existing cost models. Other issues are that these models are also hard to extend since a new model needs to be handcrafted for every new operator implementation. Moreover, today's models do not cover complex operations that allow users to call user-defined functions.

Recent approaches thus suggest to learn cost models by using DNNs instead of handcrafting them [18]. While these approaches can estimate the execution costs more accurately even for complex operations, they suffer from the general problems of using DNNs not only regarding high training cost but also explainability and robustness of DNNs plus missing update capabilities, as discussed before.

### 3.2 Fitting a Cost Model

In the following, we present a fittable cost model that combines (1) the ability of differentiable programming to encode knowledge about the general shape of cost functions for individual operators with (2) the capabilities to capture important effects of the underlying hardware by learning important parameters of the model by fitting. Figure 3 shows the basic idea of our fittable cost model.

The model is targeted towards DBMSs that execute SQL queries in a pipelined manner, which is the case for most commercial DBMSs that either implement a classical iterator model (for individual tuples or blocks of tuples) or DBMSs that rely on pipeline-based code generation for query execution, such as Hyper. In order to estimate the execution time of complete query plans, the model estimates the costs of each pipeline and then aggregates the cost to compute the total cost of that query plan. The core components of our model are thus fitted cost models that we use to estimate the costs of individual pipelines.

An important aspect is that a fitted cost model can be used to estimate the execution costs for a wide variety of different instantiations of the same type of pipeline; i.e., we learn the general behavior of a pipeline type that can be applied to different tables, rather than learning a cost model for each particular instantiation of a pipeline over a given table. For example, the cost model shown in Figure 3 (right-hand-side) can be used to estimate the costs for both build-pipelines that are executed over two different tables $S$ and $T$ by providing the features of the pipeline as input to the model. In order to enable that the same cost model can be used for different instantiations of the same pipeline type, our cost models take general features of a pipeline (such as the base table size, tuple-width, etc.) as input to estimate the execution time.

The currently supported pipeline types in our cost model are shown in Table 1 (as the first three rows). The cost model for each of these pipeline types is composed of one or multiple differentiable functions that capture the cost for each operator used in that pipeline. For example, the cost model $c_{build-pipe}$ is composed of two differentiable functions: one function $c_{scan-op}$ that captures the cost for a filter operator and one function $c_{buildht-op}$ that captures the cost of building a hash table. The fittable cost models for the operators that we use for the different pipeline types are shown in Table 1 (as the last four rows).

Another aspect of our fittable cost model is that the cost models for pipeline types, such as $c_{build-pipe}$, define weights (e.g., $w_f$ and $w_b$) that reflect the influence of a particular operator on the overall cost, when executed in that pipeline. These parameters are fitted individually for each pipeline type, since the same operator (when used in different pipeline types) can have a different influence on the overall cost. For example, the cost of materializing the output might be more dominant in a scan-pipeline than in a probe-pipeline, where the overall cost is dominated by random memory accesses resulting from probing into the hash table(s).

Finally, for the actual fitting of the cost models of the different pipeline types, we collect the actual runtime for a variety of pipeline instances for a given hardware platform. We use this collected training data for gradient-based optimization to fit the cost model and learn the parameters of pipelines end-to-end, as indicated in Table 1. The pipeline types we currently support already allow us to estimate the execution time for a wide variety of query plans ranging from simple query plans over a single table to complex query plans with multi-way hash joins over multiple tables consisting of multiple build- and probe-pipelines. In the future, we plan to extend the pipeline types to cover also other operations such as aggregations.

### 3.3 Initial Results

In the following, we show the initial results of fitting our cost model and compare the results also to recent learned cost models that purely rely on DNNs [18]. The aim of our experiments is to show that (1) white-box model can provide high accuracy for cost estimates, (2) white-box models need less training data than black-box models and (3) white-box models can generalize.

For the experiments, we implemented our fittable cost model based on the Autograd[3] framework in Python and a prototypical main-memory based execution engine in C++ to run SQL queries to collect training data. The code of our implementation is available open-source[4].

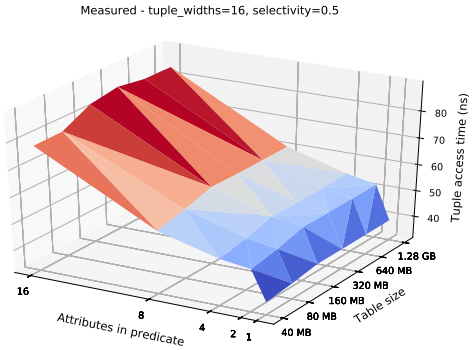
For running all experiments, we used a server with two Intel Gold 5120 Skylake CPUs (2.2 GHz, 19.25 MiB L3 cache) and 384GB of DDR4 RAM. For collecting training data, all SQL queries were executed single-threaded inside our execution engine. We make use of the Adam [7] optimizer inside the Autograd framework for fitting our cost model.

*Exp. 1 - Accuracy of Model.* In this experiment, we report the accuracy of our fittable cost model to show their potential to provide high quality estimates. To measure the quality of cost estimates in

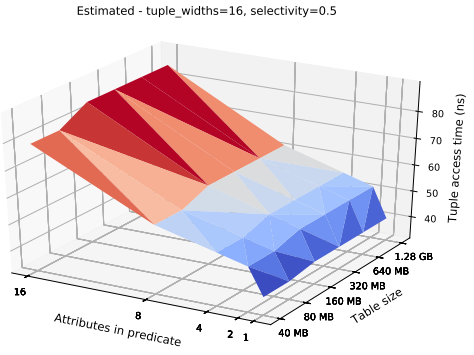
[3] https://github.com/HIPS/autograd
[4] https://github.com/DataManagementLab/cidr-cost-model/

| Name | Type | Cost function | Learned Parameters and Comments |
|---|---|---|---|
| scan-pipe | pipeline | $c_{scan-pipe} = w_f \cdot c_{scan-op}(T) + w_m \cdot c_{mat-op}(apply_{pipe}(T))$ | $w_f$ and $w_m$. |
| build-pipe | pipeline | $c_{build-pipe} = w_f \cdot c_{scan-op}(T) + w_b \cdot c_{buildht-op}(apply_{pipe}(T)$ | $w_f$ and $w_b$. |
| probe-pipe | pipeline | $c_{probe-pipe} = w_f \cdot c_{scan-op}(T) + w_p \cdot \sum_{i=1}^{n} c_{probeht-op}(HT^i, apply_{pipe}(T)) + w_m \cdot c_{mat-op}(apply_{pipe}(T))$ | $w_f$, $w_p$ and $w_m$. The pipeline can probe into multiple hash-tables denoted as $HT^i$ for the i-th join to implement multi-way joins. |
| scan-op | operator | $c_{scan-op}(T) = \begin{cases} rows(T) \cdot (a_1 \cdot \lvert T\rvert + b_1) & \lvert T\rvert < \text{L1-cache} \\ rows(T) \cdot (a_2 \cdot \lvert T\rvert + b_2) & \text{L1-cache} \leq \lvert T\rvert < \text{L2-cache} \\ rows(T) \cdot (a_3 \cdot \lvert T\rvert + b_3) & \text{L2-cache} \leq \lvert T\rvert < \text{L3-cache} \\ rows(T) \cdot (a_4 \cdot \lvert T\rvert + b_4) & \text{L3-cache} \leq \lvert T\rvert \end{cases}$ | $a_i$ and $b_i$ where $a_i$ and $b_i$ are linear combinations of tuple-width and number of attributes in the filter predicate each having its own fittable parameter. Moreover, we not only use different parameters $a_i$ and $b_i$ (i.e., segments) for different table sizes but also for selectivities $< 0.5$ and $\geq 0.5$ as well as for number of attributes in selection predicates to model effects such as branch-mispredictions and effects of cache-line sizes. However, showing the parameters for all cases in this table would decrease the readability and thus we omit them. |
| buildht-op | operator | $c_{buildht-op}(T) = w_b \cdot tw \cdot rows(T)$ | $w_b$. Cost for inserting a tuple linearly depending on tuple-width (tw). |
| probeht-op | operator | $c_{probeht-op}(HT, T) = \begin{cases} w_{p1} \cdot tw \cdot rows(T) & ht - size < \text{L1-cache} \\ w_{p2} \cdot tw \cdot rows(T) & \text{L1-cache} \leq \lvert HT\rvert < \text{L2-cache} \\ w_{p3} \cdot tw \cdot rows(T) & \text{L2-cache} \leq \lvert HT\rvert < \text{L3-cache} \\ w_{p4} \cdot tw \cdot rows(T) & \text{L3-cache} \leq \lvert HT\rvert \end{cases}$ | $w_{p1}$, $w_{p2}$, and $w_{p3}$. We use different parameters to reflect the different cost depending on the fact whether the HT fits into one level of the caches. Moreover, the cost of probing a single tuple into a HT linearly depends on the tuple-width (tw) of the probed tuple. |
| mat-op | operator | $c_{mat-op}(T) = w_m \cdot \lvert T\rvert$ | $w_m$. Cost for materializing a single tuple are constant. |

**Table 1: Fittable cost models for pipeline types and operators - $T$ is the input table of a pipeline, $|T|$ is the size of the input table in Byte and rows(T) the number of rows in T, $apply_{pipe}(T)$ is the resulting table $T$ after applying all downstream operators on $T$, $HT$ is a hash-table that is either build or probed and $|HT|$ is the size of the hash-table in Byte.**




(a) Real execution time




(b) Estimated execution time

**Figure 4: Exp. 1 - Real and estimated execution time for the scan-pipeline type for table sizes larger than L3 cache. The plots show the real and estimated execution time for the different tables sizes and number of attributes used in the selection predicate. We see that for one attribute in the selection predicate the tuple-access time is much lower since the attribute to be evaluated fits into one L1 cache line. Our cost model for the scan-pipeline captures this by using different segments in a piecewise-linear function for the filter-op as discussed in Table 1 (last column).**

| Name | Median q-error | 90th-percentile |
|---|---|---|
| scan-pipe | 1.0148 | 1.0287 |
| build-pipe | 1.0355 | 1.0663 |
| probe-pipe | 1.0403 | 1.0735 |

**Table 2: Q-error of different pipelines when trained on 100% of the training data on all table sizes. We see that the median q-error for the different pipelines is maximum** 1.0403**.**

this experiment, we use the q-error, which is the factor by which an estimate differs from the real execution time. For example, if the real execution time of a pipeline is 100ms, the estimates of 10ms or 1000ms both have a q-error of 10. Using the ratio instead of an absolute or quadratic error captures the intuition that for making optimization decisions only relative differences matter.

For collecting training and testing data, we created tables of different sizes (from 32 kB to 1.28 GB) with a varying tuple-width from 1 attribute (4 Byte) up to 16 attributes (64 Byte). For these tables we then executed query plans (single table and join queries) composed of the pipeline types supported by our cost model. In total, we thus collected the execution time for 56, 730 pipeline instances, evenly spread across the different pipeline types. Afterwards, we randomly split the data into 90% for training and 10% for testing.

The q-error (median and 90th percentile) for all table sizes are shown in Table 2. We can see that our fittable cost models can provide accurate estimates for the different pipeline types with a median q-error of less than 1.0403. While in this experiment, we used the full training data, in the next experiment (Exp. 2) we see that already 5% of the training data is enough to achieve a similarly low q-error for our model. Additionally, compared to a black-box DNN, which we also show in the next experiment, the q-error of our fitted cost model is lower and requires less training data.

We also visualize the results of the estimated costs of our model and the real execution times in Figure 4 to see that our model precisely captures the tuple access cost.

*Exp. 2 - Data Efficiency of Model.* In this experiment, we show the data efficiency of our fittable cost model. For this experiment, we use the same testing set as before but vary the size of the training data used for fitting our model. Moreover, in order to show

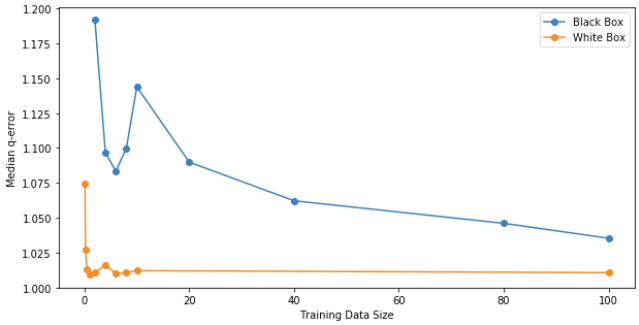



**Figure 5: Exp. 2 - Data efficiency of our fittable cost model. This plot shows the result for the scan-pipeline comparing the median q-error of our model based (white-box) to a DNN-based model (black-box) based on [18], when using only $x\%$ of the original training data.**

that our model is more data efficient than a black-box model for cost estimation, we implemented the approach suggested in [18] that uses a tree-based DNN to estimate the cost of a query plan. A tree-based DNN uses a separate DNN for each operator in a query plan that can be stacked together and trained end-to-end. Our implementation of their approach based on the Autograd framework is also available in our open-source repository.

The results for learning the cost model for simple query plans on a single table are shown in Figure 5. We can see that our white-box model can already achieve a low q-error with only 5% of the training data. In contrast, the black-box model requires much more training data to achieve a low q-error even for these simple queries. More interestingly, if we provide the full training data to the black-box model, it is not able to reach the same accuracy that our white-box model achieves with only 5% of the training data.

We also executed the same experiment for more complex query plans that include joins over two tables. The results (which we do not plot due to space restrictions in this paper) show a similar trend as for the scan-pipeline only.

*Exp. 3 - Generalizability of Model.* Finally, in the last experiment we show the capability of our cost model to generalize queries over new tables. In order to show that our fitted cost model can generalize to new unseen tables, we excluded tables sizes larger than 320MB from the training data. For testing, we used tables of sizes that the model had not seen before including table sizes that are in the range of those the model had seen before (e.g., 256MB) as well as table sizes larger than the model had seen (e.g. 512 and 1024MB). The results for estimating the cost of the scan-pipeline is depicted in Figure 6 showing the real execution time as well as the estimated execution time for these unseen tables.

As we can see, the model generalizes to these new tables. The median q-error and $90th$ percentile are similar to the results of Exp. 2. We do not show results of the black-box model that we used in Exp. 2 since this model cannot generalize to new unseen table sizes. The reason is that tables in this model are encoded using one-hot vectors; i.e., the model learns the cost estimation individually for a particular table rather than learning a cost model that is based on general features such as table-sizes as we do.

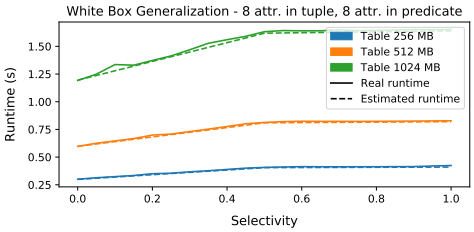



**Figure 6: Exp. 3 - Generalizability of our fittable model to unseen tables. Results show the real and estimated execution time for table sizes of** 256 MB**,** 512 MB**, and** 1 GB **that have not been used in the training set.**

## 4 CONCLUSION

In this paper, we have presented our vision towards FITable DBMSs. Based on our initial case study with a fitted cost model, we have shown that fitting not only needs much less training data but also generalizes, since the model itself captures the general shape of how the cost of operators in a DBMS typically behave.

While cost modelling is a natural candidate for fitting, we believe that fitting can be used for many other DBMS components. Furthermore, since differential programming enables end-to-end learning by composing white-box and black box models, we believe that this allows us to build holistic models that span across different DBMS components; e.g., to combine a fittable model for caching with a fittable cost model for query optimization to enable better decisions in a DBMS system.

## REFERENCES

[1] D. V. Aken et al. Automatic database management system tuning through large-scale machine learning. In *SIGMOD*, pages 1009–1024, 2017.
[2] P. A. Bernstein et al. *Principles of Transaction Processing for Systems Professionals*. Morgan Kaufmann, 1996.
[3] F. de Avila Belbute-Peres et al. End-to-end differentiable physics for learning and control. In *NIPS*, pages 7178–7189, 2018.
[4] J. Ding et al. ALEX: an updatable adaptive learned index. *CoRR*, abs/1905.08898, 2019.
[5] A. Galakatos et al. Fiting-tree: A data-aware index structure. In *SIGMOD*, pages 1189–1206, 2019.
[6] S. Idreos et al. Design continuums and the path toward self-designing key-value stores that know and learn. In *CIDR*, 2019.
[7] D. Kingma et al. Adam: A method for stochastic optimization, 2014. cite arxiv:1412.6980 Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.
[8] A. Kipf et al. Learned cardinalities: Estimating correlated joins with deep learning. In *CIDR*, 2019.
[9] T. Kraska et al. The case for learned index structures. In *SIGMOD*, pages 489–504, 2018.
[10] T. Kraska et al. Sagedb: A learned database system. In *CIDR*, 2019.
[11] V. Leis et al. How good are query optimizers, really? *PVLDB*, 9(3):204–215, 2015.
[12] T. Li et al. Differentiable programming for image processing and deep learning in halide. *ACM Trans. Graph.*, 37(4):139:1–139:13, 2018.
[13] L. Ma et al. Query-based workload forecasting for self-driving database management systems. In *SIGMOD*, pages 631–645, 2018.
[14] Q. Ma et al. Dbest: Revisiting approximate query processing engines with machine learning models. In *SIGMOD*, pages 1553–1570, 2019.
[15] R. Marcus et al. Deep reinforcement learning for join order enumeration. In *aidm@SIGMOD*, pages 3:1–3:4, 2018.
[16] R. Marcus et al. Neo: A learned query optimizer. *CoRR*, abs/1904.03711, 2019.
[17] Y. Sheng et al. Scheduling OLTP transactions via learned abort prediction. In *aidm@SIGMOD*, pages 1:1–1:8, 2019.
[18] J. Sun et al. An end-to-end learning-based cost estimator. *CoRR*, abs/1906.02560, 2019.
[19] F. Wang et al. Backpropagation with callbacks: Foundations for efficient and expressive differentiable programming. In *NIPS*, pages 10201–10212, 2018.
[20] J. Zhang et al. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *SIGMOD*, pages 415–432, 2019.