


RESEARCH

Open Access



Robust Cardinality: a novel approach for cardinality prediction in SQL queries

Francisco D. B. S. Praciano* , Paulo R. P. Amora, Italo C. Abreu, Francisco L. F. Pereira and Javam C. Machado

*Correspondence:
daniel.praciano@lsbd.ufc.br
Laboratory of Systems and
Databases, Federal University of
Ceará, Ceará, Av. Humberto Monte,
60440-593 Fortaleza, Brazil

Abstract

Background: Database Management Systems (DBMSs) use declarative language to execute queries to stored data. The DBMS defines how data will be processed and ultimately retrieved. Therefore, it must choose the best option from the different possibilities based on an estimation process. The optimization process uses estimated cardinalities to make optimization decisions, such as choosing predicate order.

Methods: In this paper, we propose Robust Cardinality, an approach to calculate cardinality estimates of query operations to guide the execution engine of the DBMSs to choose the best possible form or at least avoid the worst one. By using machine learning, instead of the current histogram heuristics, it is possible to improve these estimates; hence, leading to more efficient query execution.

Results: We perform experimental tests using PostgreSQL, comparing both estimators and a modern technique proposed in the literature. With Robust Cardinality, a lower estimation error of a batch of queries was obtained and PostgreSQL executed these queries more efficiently than when using the default estimator. We observed a 3% reduction in execution time after reducing 4 times the query estimation error.

Conclusions: From the results, it is possible to conclude that this new approach results in improvements in query processing in DBMSs, especially in the generation of cardinality estimates.

Keywords: Query optimization, Cardinality estimation, Machine learning

Introduction

Query optimization is a widely explored field of study. One of the most impactful tasks in the process of query optimization is cardinality estimation. To select an optimal execution plan among several plan alternatives, the query optimizer calculates how much the execution plan will cost. As plans are not immediately executed, the system is unable to calculate their costs in advance. Thus, it has to rely on estimates. If the estimates are off, the consequences are slow queries and performance variation.

To retrieve estimates, most commercial database management systems (DBMS) rely on histograms [1], since they allow for a trade-off in memory space to store the histogram and estimation speed, as it is very quick to get a sample from a histogram. However,

histograms are costly to build, they need to be rebuilt for refreshing and do not capture some data characteristics, such as data values correlation [2, 3].

Although many database systems can be configured to take these characteristics into account, this configuration requires expert knowledge of both the database system and the application data domain. Without this expert knowledge, this scenario leads to errors in estimation and performance hindering in systems when the workload involves complex queries and tables do have some correlation between attributes [2–5].

Also, according to Ioannidis et al. [4], cardinality estimation errors can be propagated in an exponential fashion, highly impacting the query optimizer decision.

PostgreSQL, for example, allows for that approach through the *statistics* object. It receives a set of attributes for monitoring and, as queries are executed on the table, it attempts to discover a possible correlation between the existing values in each attribute, instead of considering them separately. Then, the *analyze* command computes the statistics, through a sample of the table. It should be noted that this requires expert knowledge of both the DBMS, on how to set up these statistics, and the dataset, to use the best attributes on the value set so that the estimations are improved. Without this execution of *analyze* command, the PostgreSQL estimator is based on three principles: uniformity, independence, and principle of inclusion [2].

Machine learning techniques are currently being explored in many stages of the DBMS, from parameter tuning [6–8], to index creation and maintenance [9–11] and query optimization [12–14]. The evolving characteristic of many machine learning models makes a strong case for using these techniques to model problems without a high cost to update and obtain good results as more instances are observed.

By using machine learning models, it can be possible to achieve lower estimation errors and better execution times without having all the expert knowledge needed to do so, therefore, improving the general performance of the DBMS.

This work proposes Robust Cardinality, a machine learning backed approach to estimate query cardinalities with a given confidence, being able to judge whether its estimations are good enough. This technique is said to be robust since it is possible to manage the prediction's uncertainty. We implemented Robust Cardinality in PostgreSQL and obtained improvements in both error estimation and execution time. We also compared estimates with a state-of-the-art machine learning estimator, obtaining a majority of better results. This work's contributions are as follows:

Efficient machine learning technique to predict query cardinality. Using machine learning models enables us to achieve better cardinality estimations for SQL queries, resulting in better cost estimation and better query plan choices. Our approach quickly builds models with very low overhead on the query execution time differently from other approaches based on neural networks. Simultaneously, considering 90-percentile of estimates' accuracy through the Q-error metric, Robust Cardinality has 4x lower error than the traditional technique based on histogram and 2x when compared with a modern technique based on machine learning using the same dataset. Regarding the query execution time, our approach can reduce the query runtime by around 3% in comparison to the default PostgreSQL implementation.

Evolving model construction. The process of building a model is continuously evolving as data is updated and query cardinality changes. Robust Cardinality

accumulates previous knowledge to build new models; therefore, it constantly adapts to query data evolution. None of our competitors presents an evolutionary approach or keeps the metadata out of date as in more traditional approaches.

Robust approach to inaccurate predictions. Robust Cardinality supports a confidence interval on query cardinality prediction. During estimation, if the prediction seems to be inaccurate, an optimized execute plan found by the traditional technique is scheduled for execution on the query engine. Thus, Robust Cardinality is more flexible than recent related works that are also machine learning-based techniques.

Real DBMS experimental results. We have reported results from experiments using a recent implementation of the PostgreSQL query engine. Although we could have experimented independently from a relational DBMS as most of the other approaches, we wanted to evaluate Robust Cardinality in an established and more than approved DBMS with a real SQL query engine. In addition, we were able to analyze our approach against traditional histogram techniques and make use of them whenever our predictions were outside of a confidence interval.

The paper is organized as follows: The “[Related work](#)” section discusses related works. The “[Robust Cardinality](#)” section explains how Robust Cardinality is designed, highlighting how the estimates are generated over time and how it integrates with a DBMS. The “[Experimental evaluation](#)” section presents the experimental evaluation, and, finally, we conclude in the “[Conclusion and future work](#)” section presenting our final remarks and possible future works.

Related work

Cardinality estimation is one of the problems of query processing that most gained attention in the database scientific community because of its importance in the whole database systems stack and performance, see recent surveys, such as [15, 16] for an overview. Histograms [1] and sampling techniques [17] were the first two techniques proposed to solve this problem. The latter was not adopted by the main systems because of its intrinsic high cost, despite its good accuracy. In contrast, most relational database systems implemented estimation techniques based on histograms using some premises to deal with all kinds of queries. Although its accuracy is good for most cases, there are some cases where this technique does not fit at all. For example, the histogram is not suitable when there is too much correlated data, because usually it is uni-dimensional, therefore, does not capture correlation data between attributes.

In an attempt to overcome the cases where histogram-based techniques have a bad performance, some works in the literature [18, 19] propose to incorporate a feedback cardinality loop, a method that monitors the cardinalities present in the query plans at run time with the purpose of obtaining the true cardinalities and, consequently, injecting them in the next queries that involve the cardinalities already acquired. Exploring this valuable knowledge obtained through the queries already executed brings to light the possible errors that occurred in the process of optimizing these queries, enabling the identification of the part where an inaccuracy of the histograms’ estimates occurs.

Liu et al. [20] propose the use of machine learning techniques to address the cardinality estimation problem. The approach consists of using neural networks to learn selectivity from SQL query predicates. In this way, they model the problem as a supervised learning

one which replaces the histogram with a neural network. This approach only allows for single table queries. Thus, several practical and realistic queries are not addressed by it.

Since then, many works choose to dive into the subject [21]. Dutt et al. [22] present a cardinality estimator in a similar approach to Liu et al. Instead of only neural networks, they use regression machine learning models, such as ensemble trees. This approach makes all predicates have a selectivity estimate. To do so, they model queries in a way that if an attribute is not present in a predicate of the query, a predicate bounded by upper and lower values is created only as a filler, since the model requires queries with all attributes as predicates. As in this work, the existing technique applied in the DBMS is not excluded, orchestrating the prediction work with the DBMS. In contrast, our work considers the joining clauses as well that are not addressed in their work.

Kipf et al. propose learned cardinalities [23], a technique that uses a deep learning approach formed by a neural network set named MSCN [24]. Queries are split into 3 sets that are input in MSCN, a table set, a join set, and a predicate set. Each of these sets is encoded into one-hot vectors. After these sets are built, each of them is input to a neural network, where the outputs are averaged through pooling, resulting in a compact representation of the query. This is used as input to a final neural network that, from this input, generates the query cardinality estimate. An interesting contribution from that work is a query generator, intended to create a labeled training set to the prediction model. In summary, the algorithm generates uniform random numbers to select the tables and predicates in a query, then executes the query against the database to obtain its real cardinality, then, both query and cardinality are added to the training set.

Woltmann et al. [25] change the previous work on the neural networks. Instead of having a model that oversees the entire database, it constructs a single neural network for each sub-part of the database, handling it as a local approach, not a global one. This changes the query modeling and incurs in an extra decision for the optimizer, which models should be used to estimate cardinality. Then, several models can be used when generating the query cardinality estimate.

Negi et al. [26] tackle the cardinality estimation problem from a different perspective. Instead of focusing on minimizing the general cardinality estimation error, it takes a more direct approach in checking whether the generated plan had a better execution time than the default through a metric called *plan-error*. As a consequence, the learned cardinality estimators are penalized according to execution time rather than estimation error. It shows that, although aggregates of estimate errors are higher, execution times are lower, because the estimator is more accurate in the nodes that impact most in the execution times.

We use learned cardinalities as a baseline in this work. Our query encoding technique is also inspired by this work, although with minor differences, for example, we chose to have a single vector instead of three sets, and there is not a sampling related section in the vector, because sampling is not used. In our work, we use a gradient boosting decision tree (GBDT) model [27, 28] instead of a neural network, as it has advantages in training time and a way to increment new information to the model, providing the ability to update it periodically, allowing for an evolutionary process. Besides that, we add a measure of uncertainty into the processes, so that the machine learning model can feedback how close its estimations are to the real cardinality. Therefore, the database system can choose

between using that estimation or to make one using a standard method, histogram, for example.

The related works highlight two open lines of research, which are addressed in this work: uncertainty handling in the estimates and integration with a real system. While this is mentioned in Dutt et al., the experiments in these related works are focused on the cardinality estimates and do not perform end-to-end testing using a full-fledged system. Also, they use predictions as they come, not allowing for expert knowledge to ponder whether these predictions can be improved.

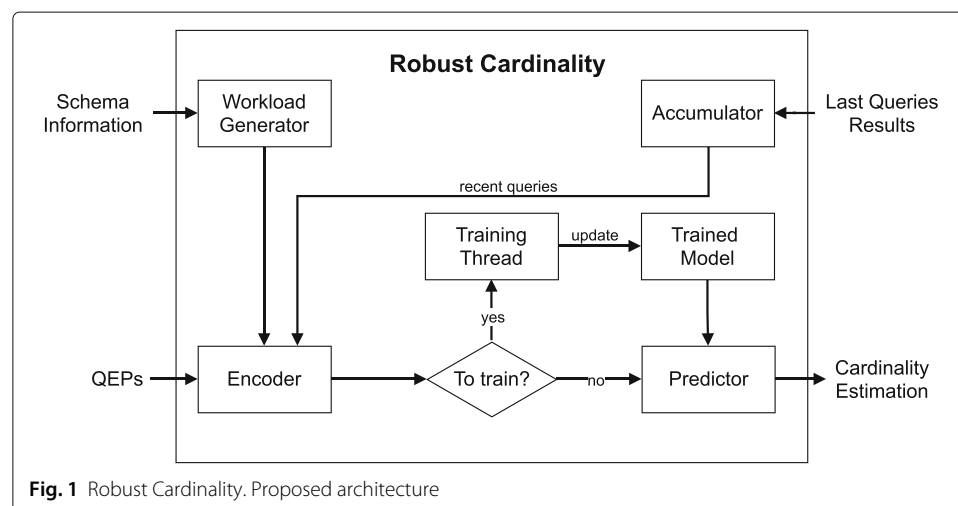
Robust Cardinality

In this paper, we propose Robust Cardinality, a cardinality estimation model using machine learning that can be periodically updated. It can be used in an evolving environment, where data is still added, updated, and read. We use GBDT as the machine learning algorithm, since it combines a number of weak learners to create a complete and strong model and allows for controlling the uncertainty of the answer. This machine learning technique has been widely used in many types of learning tasks due to its accuracy as well as efficiency. The “[GBDT in cardinality prediction](#)” section explains how this algorithm works in more details and how to apply it in our scenario.

This machine learning technique is specially adequate for our scenario because the computational cost and taken time for training and refreshing are low when compared with other strategies, such as neural networks or deep learning models, and stays sufficiently efficient. In this way, we can retrain the model periodically to capture database changes with easiness. This technique also allows calculating the uncertainty of the model output, which means that we can know how confident is the model in the calculated response, enabling it to opt whether to use it or not.

Our model operates in the query processing and optimization phases. Its architecture is shown in Fig. 1. The model communicates with several components of the DBMS. At the workload generator, a randomized workload can be generated for the initial training in a way similar to [23]. It is important to highlight that any method for generating queries can be used.

This module communicates with the catalog to obtain schema information to generate the workload. The encoder module processes and transforms queries into input for the

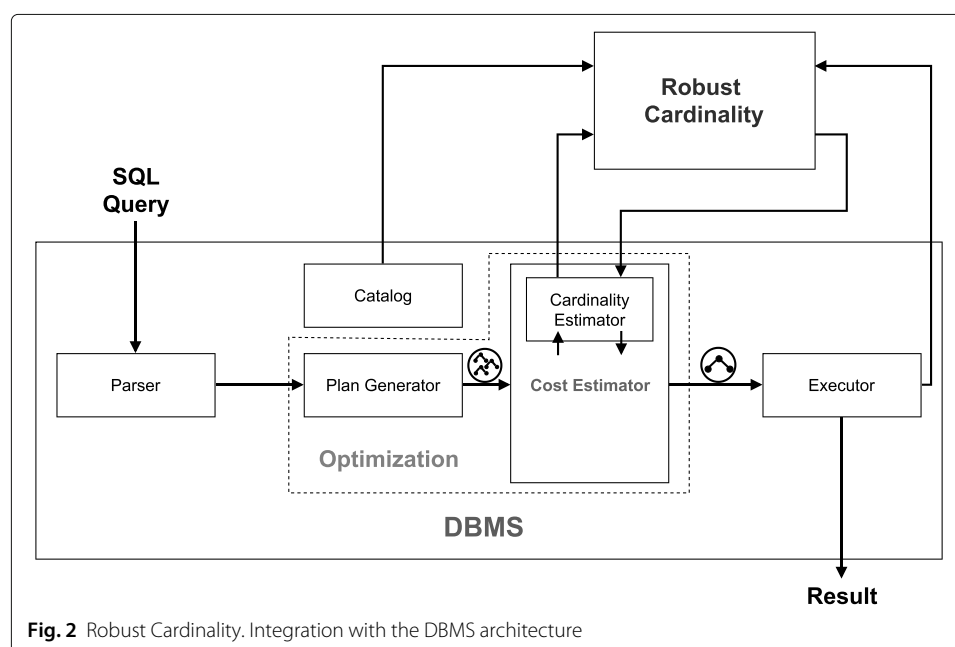


model. Then, a decision is made whether the model must be refreshed or not. If so, the training thread retrieves encoded data and updates the model, else, the query is passed on to the predictor module, which uses the generated machine learning model to predict the query cardinality estimates. After the query is processed and there are real values for the cardinality, these are stored in the accumulator and will be used once the decision to refresh the model is positive.

In Fig. 2, the DBMS integration is shown. It shows the communication with the catalog and with other DBMS components. The query is preprocessed before it can be analyzed in the optimization process. During the query optimization, the DBMS explores many of the possible plan trees in the plan generator. To estimate the cost of each one of them, the cardinality estimator will use the model to calculate each plan node's cardinality, providing, as input to the model, a representation of these. The plan with the lowest estimated cost will be executed and, after execution ends, the model gets query results from the executor and stores them for its updating process in the future.

The technique allows for a non-invasive model integration, where some components' output is used as input for the model, such as retrieving table schemas from the catalog, reading current estimates from the estimator, and retrieving the actual cardinalities after the query is executed. Meanwhile, the model updates the estimates in the cardinality estimator using its output, to allow the optimizer to choose the best plan based on more accurate estimates. There may be times where the model's estimates are far off, for example, when there is a sudden change in the workload. We discuss later on how to handle inadequate estimates.

The technique is designed as plug-and-play, requiring a minimal change in the DBMS or none, if there is an interface that allows the model to modify the values of the cardinality estimator. This also allows it to fall back to the DBMS native method of estimating cardinalities, in case the model is not confident enough in its estimates or more specific situation, such as selections from a base table, where the histogram can perform well.



GBDT in cardinality prediction

GBDT is an ensemble supervised learning algorithm. In this approach, a set of learners, namely decision trees are trained to learn the queries' cardinalities in order that each one of these learners generates a prediction. These predictions are combined into a single final prediction to obtain the query cardinality's estimates using some strategy, for instance, mean value [29, 30]. Figure 3 shows the architecture of this algorithm

In GBDT, a learner is said to be weak when its predictions are slightly better than those of a random one. The motivation behind ensemble methods is that the combination of the predictions of several weak learners generates a robust, powerful, more representative model [32]. Lately, this technique has shown good predictive power in different types of learning, such as regression [33], which is the type of task Robust Cardinality is modeled after. Although there are many ensemble supervised learning algorithms, such as bagging, boosting, stacking [34–37], we chose GBDT, a boosting-based algorithm, due to the fact that this technique allows to use the quantile regression technique [38] permitting to manage the uncertainty of the model predictions. This is a strong point of our work since the DBA will be able to choose a parameter that will be used to decide if it uses the predictions of our model or falls under a default approach.

Training a model using GBDT in Robust Cardinality requires a training set X composed of encoded queries and labels of their respective cardinalities y . Details about the query encoding process will be explained in the following subsection. This set will be used to specialize the structure of the learners through the boosting technique [39] in which each of the learners is obtained one after the other with the most current learner being trained taking into account the errors that occurred in the learning process of the previous ones. Formally, this process can be resumed in the following equation.

$$\begin{aligned}
 M_k(X) &= \underbrace{M_{k-1}(X)}_{\text{current model}} + \underbrace{\lambda \gamma_k E_k(X)}_{\text{new learner}} \\
 &= M_{k-1}(X) + \arg \min_{\gamma_k, E_k} \sum_{i=1}^n \underbrace{L(y_i, M_{k-1}(x_i) + \gamma_k E_k(x_i))}_{\text{cost function}}
 \end{aligned} \quad (1)$$

To obtain the model M_k composed by the k already trained learners, the current model M_{k-1} is used alongside the new learner E_k , parameterized by two factors: learning rate λ and contribution factor γ_k . While λ is provided by the user, γ_k is adjusted by the GBDT in order to calculate the influence that the new learner E_k will have in the final model M_k . Finally, a differentiable cost function L is used to find the optimal splits of the new learner's nodes, i.e., that one that will minimize the overall cost. In this work, we chose

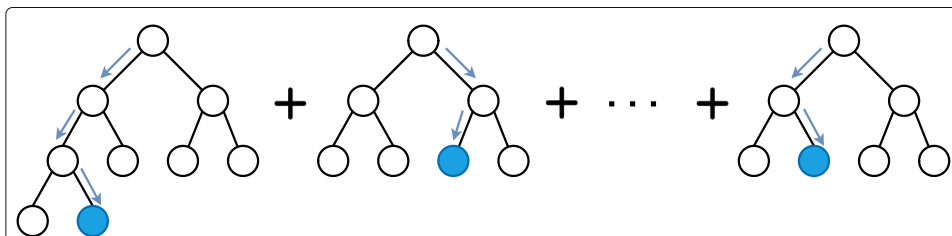


Fig. 3 An example of GBDT's learners [31]. Each of one is a decision tree that was trained to generate a good prediction illustrated by the blue nodes. All of these predictions are then transformed into a single final prediction through a chosen function, commonly mean or voting. Hence, this final prediction will be used as estimates by the optimizer

the Q-error metric (defined in Eq. 2) as our cost function so that the query cardinality estimates' errors are decreased.

In this work, we used a new implementation of GBDT that incorporates two techniques, namely, the gradient-based one-side sampling (GOSS) and the exclusive feature bundling (EFB), in order to reduce the overhead of training a model when dealing with huge datasets.

GOSS is a technique that retains instances with large gradients while performing random sampling on instances with small gradients, on the assumption that instances with small gradients present small training error, and those with large gradients present worst ones.

EFB merges data features that are mutually exclusive to reduce training complexity. This reduces the number of features that must be processed by the model, reducing the dimensionality of data. It can dramatically reduce the complexity of the algorithm when working with high dimensional data.

Encoding

To use GBDT, the SQL query nor the structures such as abstract syntax trees and plan trees can be consumed as-is. Therefore, the queries must be encoded, and this procedure is shown in Fig. 4. The query is encoded in a vector, split into sections, in a way that each table and predicate is represented. Predicate values must be numeric and are normalized in min-max fashion. In the figure, this is represented by the vector x . Indexes $x[0]$ to $x[2]$ encode the tables in the database that are being queried, where 1 means this table is in the query, and 0 means it is not. Indexes $x[3]$ to $x[12]$ represent the non-join predicates, they are split in a one-hot encoding vector for the attributes ($x[3]$ to $x[8]$), a one-hot encoding vector for the operator ($x[9]$ to $x[11]$) and an entry for the predicate value ($x[12]$). Indexes $x[13]$ to $x[15]$ are added when there are joins, as a one-hot vector where each entry is a possible attribute join. In this example, the join between $D.id_dep$ and $E.id_dep$.

To be used as a label, cardinality goes through two transformations. The first is the logarithmic transformation, which is the application of the *logarithmic* function in the cardinality and a min-max normalization similar to that used for numeric attributes. In other words, the model will receive as input the queries' cardinality normalized by these two transformations. In this way, it will also produce predictions of normalized cardi-

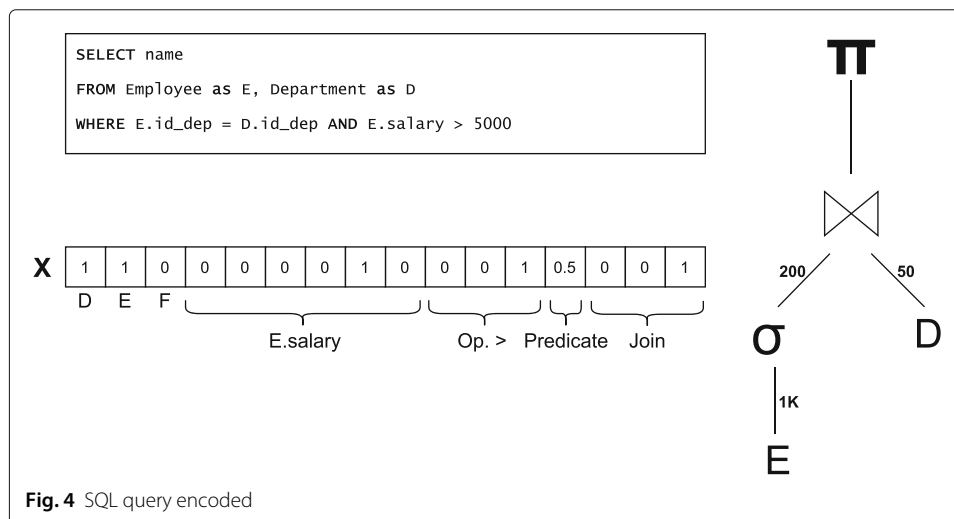


Fig. 4 SQL query encoded

nality. To obtain the cardinality estimate, two transformations that invert the operations performed to normalize the cardinality are applied. First, a transformation is applied to undo the min-max normalization and, subsequently, the logarithmic transformation is undone by applying the *exponential* function to the value obtained immediately after undoing the min-max transformation. With that, we obtain the cardinality estimate.

Training and evolution

Model training and evolution are offline operations from the DBMS perspective and happen asynchronously. The dataset is composed of the encoded queries and their label, i.e., the cardinality. As said above, the original cardinalities are also preprocessed, being subjected to two transformations, in which a logarithmic function is applied and a normalization of the values is made, using min-max. When a prediction is made, the reverse operations are applied, to retrieve a real value for the estimate.

To perform the initial training, a dataset composed of encoded queries and their cardinalities is needed. Therefore, we can either have a set of previously executed queries, alongside their real cardinalities or generate a synthetic dataset based on the database schemas. The latter can be achieved by an algorithm similar to Kipf et al. [23]. However, in our work, we vary the number of joins from 0 to 4, instead of the original that considers only up to 2 joins. It is important to note that this number of joins is a parameter that can be chosen according to the DBA. Furthermore, by increasing the number of joins in the queries that can be generated, it is possible that a greater variety of queries is obtained, consequently the dimensional space of the queries' features is better covered. Thus, the trained model is more likely to yield a better generalization for the cardinality estimates. This is one of the possible ways to deal with the well-known problem of the curse of dimensionality that is common in machine learning.

The training algorithm is shown in Algorithm 1. In summary, it generates uniform random numbers to make the choice of relations and the selection and join predicates present in a query, executes this generated query in order to obtain its cardinality and then adds it to the training set that is being generated. Finally, it returns this training set X labeled to be used in training the machine learning model.

This type of training allows for using the technique from start, instead of having to wait for queries to execute without our input. This initial training set only has the query final cardinality, not the inner operator cardinalities. This observation is important because more joins means more error propagation from one operator node to another, that is, a difference between the real and the estimated cardinality of an operator node in the query tree may propagate exponentially and cause massive differences in estimations up the tree [4]. Our model predicts the output of the query, i.e., the root of the plan tree. To avoid only having the final cardinality, the estimate is split along the tree, being recursively computed for each child node except the leaves. Each child node corresponds to an intermediate operation of the plan. To estimate the leaves' cardinality, we make use of the DBMS stock method to estimate cardinalities, usually a histogram. This technique is very effective for single node operators, which is the case of the leaves, and avoids more overhead. In this way, we can mitigate estimate errors while they do not outgrow the original values. It is also important to notice that, due to the nature of the technique, neighbor nodes do not influence one another, since they are not inputted concurrently into the GBDT.

Algorithm 1: Training Set Generator. Based on [23].

Input: Relation set \mathcal{R} , number of queries n **Output:** Labeled training set X

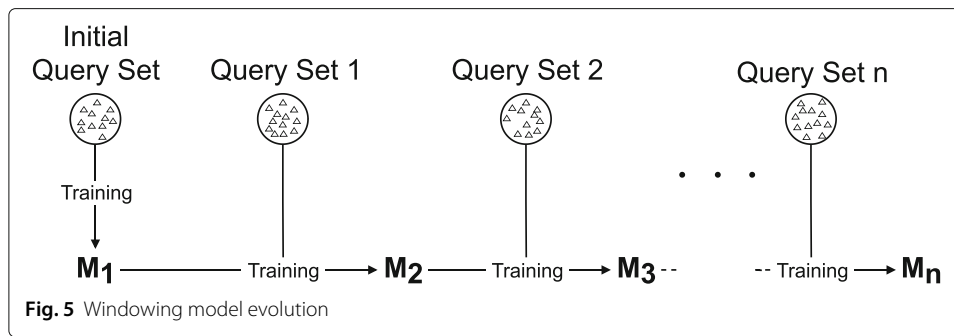
```

1 begin
2    $X \leftarrow \emptyset$ 
3   for  $i = 1 \rightarrow i = n$  do
4      $R_0 \leftarrow$  Choose randomly a relation that is referenced by others on set  $\mathcal{R}$ 
5      $\mathcal{R}_i \leftarrow R_0$ 
6      $N_{\mathcal{J}} \leftarrow$  Choose randomly a number of joins in between 0 and 4 for the new
       query  $Q_i$ 
7      $\mathcal{J}_i \leftarrow \emptyset$ 
8     for  $j = 1 \rightarrow j \leq N_{\mathcal{J}}$  do
9        $R_j \leftarrow$  Choose randomly a relation that is referenced by at least one
       relation on set  $\mathcal{R}_i$ 
10       $\mathcal{R}_i \leftarrow \mathcal{R}_i + R_j$ 
11       $\mathcal{J}_i \leftarrow \mathcal{J}_i +$  join condition between  $R_j$  and  $\mathcal{R}_i$ 
12    end
13     $\mathcal{P}_i \leftarrow \emptyset$ 
14    for  $j = 1 \rightarrow j \leq |\mathcal{R}_i|$  do
15       $N_{\mathcal{P}} \leftarrow$  Choose randomly a number of predicates between 0 and the
       number of non-key attributes on relation  $\mathcal{R}_i[j]$ 
16      for  $k = 1 \rightarrow k \leq N_{\mathcal{P}}$  do
17         $P_k \leftarrow$  Choose randomly an non-key attribute of  $\mathcal{R}_i[j]$ , an
       mathematical operator and constant values to make a selection
       predicate
18         $\mathcal{P}_i \leftarrow \mathcal{P}_i + P_k$ 
19      end
20    end
21     $Q_i \leftarrow$  Make a new query using  $\mathcal{R}_i$ ,  $\mathcal{J}_i$  and  $\mathcal{P}_i$ 
22     $c \leftarrow$  Run query  $Q_i$  and obtain its cardinality
23     $X \leftarrow X + (Q_i, c)$ 
24  end
25  return  $X$ 
26 end

```

Model evolution is achieved by using an observation window, in which W queries are stored, forming a *Query Set*, then used to improve the existing model. Figure 5 illustrates this process. The first k queries correspond to the initial training, then, when the window reaches its limit, a decision to update the model is made. While the updated model is being generated, the current model stays active, since it is an offline process. When it is ready, the current model M_1 is replaced by the updated model M_2 .

To generate M_2 , the queries in the window are used to update M_1 , in this way, the previous knowledge is not lost and the update process also does not take as long as the initial training. The model also does not grow much in size because, when the update



happens, the previous model is summarized then updated. Therefore, the complete history of the model is not stored, instead, a summary of previous evolutions alongside the current update. This process is repeated until a *Query Set n* is reached. The number of times n can be defined as much as necessary to obtain an accurate final model trained.

Uncertainty

As shown by Leis et al. [2], errors in cardinality estimation impact greatly on the DBMS performance. When using machine learning models to predict cardinality values, it is a good practice to embed a form to evaluate how confident are these predictions, taking into account that it is not always possible to cover the search space in the training set. This is known as the curse of dimensionality and it is a well-known problem in the context of machine learning. The predictions' uncertainty management makes the machine learning models more reliable and allows for identifying the cases where their use can guarantee higher accuracy estimates.

In our approach, we manage uncertainty through quantile regression technique, used on GBDT altering the loss function. This approach is not directly translatable for other techniques, as stated by Kipf et al. [23].

Uncertainty is a useful metric to avoid errors by the model, in this way, we calculate it from a prediction interval, which contains the real value in a known probability, i.e., an error margin. Alongside the cardinality estimation model, we implement two quantile prediction models that calculate upper and lower quantiles of this interval. The confidence level is the difference between the quantiles, which means, to have a 90% confidence level, both the 5% and 95% quantiles must be calculated. Given a new query, the cardinality estimates will be calculated using the current model and check if they are trustworthy by calculating the quantiles using the others models. This uncertainty can be set by a DBA, which allows for higher control and usage of expert knowledge.

By this approach, if the prediction is off the interval or if the difference between the quantiles real value is very high, there is a very high probability of the prediction being off. When this happens, the cardinality estimates generated by the model are highly penalized, allowing for the DBMS to ignore them and use the default approach.

After the encoded query has its operator cardinalities estimated, Robust Cardinality updates the values in the estimator, allowing for the optimizer to better select the execution plan among the candidates because now the estimates are closer to the actual cost.

Experimental evaluation

Setup and benchmark

The experiments were executed with an Intel Xeon E5-2609 v3, with six physical cores, 1.9 GHz per core, with an Ubuntu OS, with kernel version 4.15.0. Robust Cardinality is implemented in Python 3, using libraries *Scipy* [40], *scikit-learn* [41], and *LightGBM* [42] to implement the GBDT algorithm. For the DBMS, we used PostgreSQL 12 stable, as it is the most recent version.

We modified PostgreSQL to be able to obtain the execution plans and also to inject the new cardinality estimates into it. To store the queries and executions, a hidden table was created within PostgreSQL, which stores the query and the actual cardinality. The new estimates injection is done by modifying PostgreSQL structures named *paths*, which are simple representations of query plans that are not directly executable. After evaluating the *paths*, the DBMS expands the best of them into an actual execution plan. No other modifications were made; so, any other PostgreSQL features that allow for a better cardinality estimation by default were left untouched.

To evaluate Robust Cardinality, we used the Internet Movie Database (IMDb) dataset. IMDb stores data about movies, actors, directors, and their relationships and comprises 21 tables. According to Leis et al. [43], given the characteristics of this database, such as correlated attributes, non-uniform distributions, among others, cardinality estimation is a harder task for DBMSs. Concerning the workloads, we use three: synthetic, scale, and JOB-light. The synthetic one is composed by 100,000 synthetic queries that were generated using the process described in the “Robust Cardinality” section, 90% used to train the model, 5% to validate, and 5% to test. The queries are select-project-join queries. The other two workloads were proposed in [23]. While the workload scale is also synthetic, JOB-light is composed of 70 queries from a real workload known as join order benchmark (JOB) [2] based on the IMDb dataset. As proposed, the scale has 500 queries. However, when we re-execute these queries to obtain the true cardinalities in our recent dataset version; one of these queries had five relations on joint operations. Moreover, as these five relations have a large number of tuples, it was not possible to re-execute it, as a full memory error has always occurred, even increasing the size of the memory destined for PostgreSQL. Thus, we leave this query out, making the scale composed of 499 queries. The JOB-light, on the other hand, was successfully re-executed, allowing the 70 queries to be used for our evaluation.

Regularization and hyper-parameter tuning

We used 90000 synthetic queries to conduct model training. A well-known problem in the machine learning literature is the problem of over-training, which can lead to over-fitting. That is, as we are training our model with a large number of queries, there is a need to verify that the model will not be very well trained to answer only those queries, while its power of generalization will be low for queries not seen in the training. To avoid this, we apply four approaches that are used to prevent the generated model from being biased. The first is that we carry out an L2-regularization with a factor of 0.33 that adds a term in the loss function to be optimized, preventing the generated model from being well adjusted for training queries. Another approach is that we set up a minimum of five training queries in such a way that each leaf node of the generated decision trees must have. Thus, the decision trees will not be as deep, therefore not adjusting too much to

Table 1 Hyper-parameters and their values

Hyper-parameter	Evaluated values
<i>max_depth</i>	8; 16
<i>num_leaves</i>	32; 64; 128
<i>n_estimators</i>	1000; 5000; 10,000
<i>learning_rate</i>	0.01; 0.1

the training data. Finally, the third and fourth approaches are interconnected. Basically, they limit the number of data that will be used at each stage of the training. In the third, only 90% of training queries are chosen at random to be used in the construction of each decision tree. Similarly, the fourth strategy limits the training process to use only 90% of the features of each query to build the model.

Furthermore, we used 5000 synthetic queries to tune the configuration of hyper-parameters that LightGBM has for the model. To reduce our search space, four of them were selected. Table 1 shows the selected parameters and possible values. With the evaluation of several combinations, the one that presented the lowest error overall was: {*max_depth*: 16, *num_leaves*: 64, *n_estimators*: 10000, *learning_rate*: 0.01}.

Error is calculated through the *Q-error* metric, which represents how far the predicted values are from the true ones. This metric is suitable for our scenario because we are interested in the relative impact factor that cardinality errors can have on query processing, as stated by [5]. Formally, *Q-error* metric is presented in the following equation, where c and \hat{c} represent the true cardinality and the predicted one, respectively:

$$Q - error(c, \hat{c}) = \frac{\max(c, \hat{c})}{\min(c, \hat{c})} \quad (2)$$

For a *Q-error* e , we have that $e \in [1, +\infty]$, since the values c and \hat{c} are always positive. Note that $e = 1$ occurs whenever $\max(c, \hat{c}) = \min(c, \hat{c})$, which implies $c = \hat{c}$, that is, the estimate is equal to the real value of the cardinality. Therefore, $e = 1$ is optimal, since the predicted and true values are equals in this case, i.e., the prediction is correct. In summary, the closer the e is to 1, the closer c and \hat{c} are. Consequently, a lower *Q-error* is better.

Accuracy and uncertainty management

To verify our first contribution, we tested the Robust Cardinality model against the MSCN model proposed by Learned Cardinalities [23] and the model based on the histogram technique that comes with PostgreSQL. After training both models and preparing all of PostgreSQL's statistics, the same synthetic workload was tested against the three techniques. Table 2 presents the aggregated metrics for each one.

MSCN has the lowest mean and the lowest maximum; however, as we can see from the median and percentiles, Robust Cardinality achieves a lower error in most of the queries,

Table 2 Aggregated *Q-error* of cardinality estimates of synthetic workload by each model (lower is better)

Technique	Mean	Median	Percentile			Maximum
			90	95	99	
PostgreSQL	171.120	1.775	20.352	80.000	964.000	314,187.000
MSCN	7.659	2.000	9.829	20.618	77.337	6,586.000
Robust Cardinality	21.920	1.243	5.000	12.245	50.690	32,803.500

as shown by the median and percentiles. For comparison, Robust Cardinality's 40th percentile (1.14) is lower than MSCN's 10th-percentile (1.15), which means that the Q-error of 40% of the estimates of Robust Cardinality is lower than the error of 10% of the estimates of MSCN. Robust Cardinality's 90% percentile also matches MSCN 80%, both at the value of 5.00, meaning that we provide a higher number of estimates with errors below a given value. On the other hand, as can be seen in the table, Robust Cardinality is much more sensitive to some queries that can be considered as outliers, given that in most of the workload the model produced good estimates of cardinality. This sensitivity is demonstrated in the maximum error values obtained by each technique. Our mean is higher due to a few outliers. This few queries are characterized by selecting over thousands of tuples and returning only 0 or 1 tuple. This sensitivity is due to the normalization process applied to queries' cardinality. As stated in the "Encoding" section, the exponential function is applied to the model's output to obtain the cardinality estimate. Therefore, any small error that the model may present can have an exponential impact on the cardinality estimate. As we will see below, this type of queries can be controlled by our model through uncertainty management.

To manage the uncertainty, we have defined a threshold that will quantify the level of uncertainty of the queries that will be accepted to use the cardinality estimates of our model. The queries that will be rejected mean that the estimates given by our model have a high level of uncertainty, and therefore, it may be a better option to execute them using a standard technique such as histograms. As explained in the "Uncertainty" section, we have, for each query, a lower and upper bound of the normalized cardinality estimate. Furthermore, the greater the interval defined by these two values, the greater the uncertainty linked to the cardinality estimate. In this way, the threshold is used to limit this range and, consequently, it limits the uncertainty and also the likely queries' Q-error that will use our cardinality estimates. Lastly, expert knowledge can be applied by regulating the threshold.

For the evaluation of how the uncertainty management impacts the estimates, we performed an experiment varying this acceptance threshold. As stated above, the threshold is related to the upper and lower bounds of the normalized cardinality estimate, and it is compared to the value obtained by subtracting the lower bound from the upper. These two values are normalized; therefore, a threshold of 0.0 would accept only queries where the lower bound is exactly equal to the upper one (i.e., there is a certainty that the predicted normalized cardinality value is equal to the true one), and a threshold of 1.0 would accept all queries since it is the maximum possible difference. In summary, the lower the threshold, the lower the confidence interval tolerance for cardinality estimates. The threshold was varied from 0.1 to 0.5. On the one hand, a threshold of 0.1 signifies that only queries with high confidence in their estimates are accepted, i.e., the upper and lower bounds difference is less than or equal to 0.1. It restricts the uncertainty about the cardinality estimates for the accepted queries. On the other hand, a threshold of 0.5 represents an acceptable difference of 0.5 between the upper and lower bounds normalized values, a very high tolerance interval and, consequently, more uncertainty is allowed. The results can be seen at Table 3.

Each column corresponds to a threshold value, i.e., the error tolerance for query estimates. Second row shows the number of accepted and rejected queries for each threshold value and last row, the 99th percentile of Q-error. Increasing the threshold means having the model to accept estimations in which it has less confidence. That is why the bigger

Table 3 Synthetic queries accepted and rejected and their Q-errors for each threshold value

Threshold	0.1		0.2		0.3		0.4		0.5	
# of queries	Acc	Rej	Acc	Rej	Acc	Rej	Acc	Rej	Acc	Rej
99% Q-error	3.3	59.0	9.4	90.8	16.8	170.1	43.4	292.5	54.9	157.2

the threshold, the more queries are accepted. It is also possible to verify that even with a high threshold of 0.5, the 99th percentile Q-error for the accepted queries is lower than the rejected ones. It means the predictions for accepted queries are very accurate.

Figure 6 shows the amount of error for each query. As corroborated by Table 2, except for a few outliers, almost all queries have their errors close to 1. Two of the queries have an Q-error over 32000, being the cause of the elevated mean. These queries were removed from the figure above because of the interval error would be very large; hence, it would not be possible to get a close view of the errors. Furthermore, uncertainty management is also shown in this image through the colors of each of the points that represent the queries. Note that blue tones represent queries with a more restricted threshold, allowing little uncertainty in cardinality estimates. The points whose color is light blue are close to 1, which shows that the accepted queries with a threshold equal to 0.1 obtained cardinality estimates quite accurate by our model.

Using the same model trained by the synthetic workload, we investigate the results obtained by the other two workloads. Initially, we analyze the results obtained in the execution of the scale workload. Looking at Table 4, it is possible to see that Robust

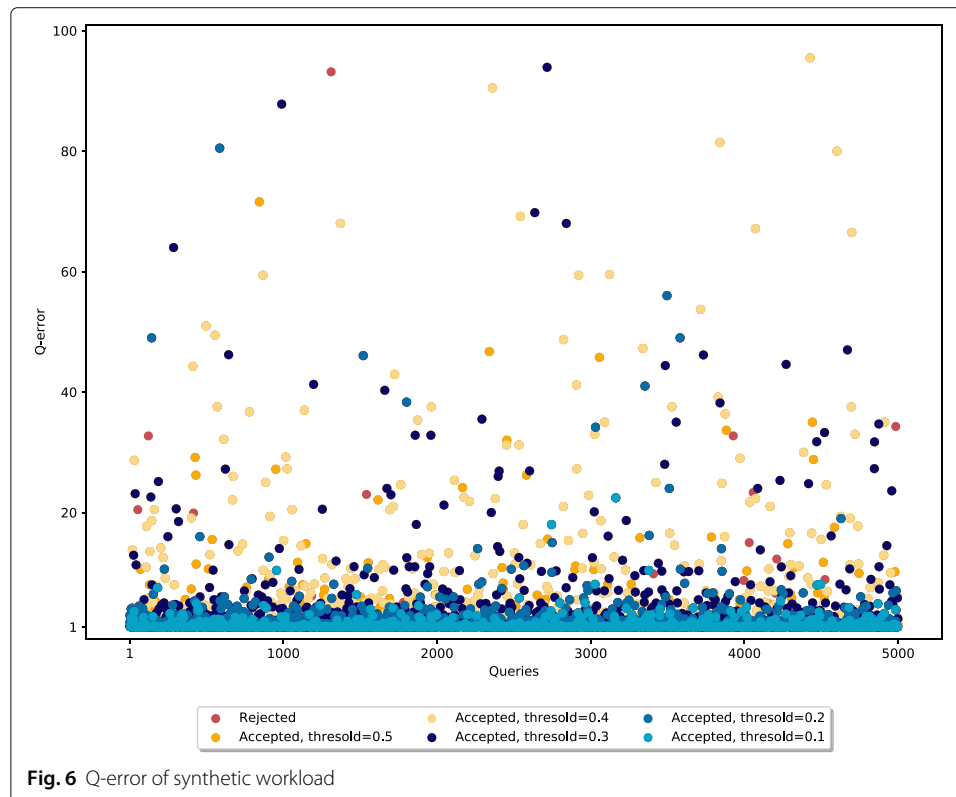


Fig. 6 Q-error of synthetic workload

Table 4 Aggregated Q-error of cardinality estimates of scale workload by each model (lower is better)

Technique	Mean	Median	Percentile			Maximum
			90	95	99	
PostgreSQL	530.244	2.682	132.972	393.228	1,641.207	223,040.000
MSCN	31.005	3.211	41.068	96.326	601.186	2,221.609
Robust Cardinality	78.431	1.371	37.638	91.794	601.735	25,943.000

Cardinality's estimates are still lower than that of MSCN in this workload as well, especially the median, 90th and 95th percentile. It can be observed that our model showed a median more than twice as low as that of the MSCN, which implies that our model generated much more precisely cardinality estimates. In addition, the 99th percentile of both techniques are almost equal, around 601, differing in decimal places. Therefore, it can be concluded that Robust Cardinality is generating more accurate cardinality estimates. However, it occurs again that the MSCN is less sensitive to those queries whose cardinality is small. This is seen in the analysis of the maximum Q-error values, which in the MSCN is approximately 10 times less than that of Robust Cardinality.

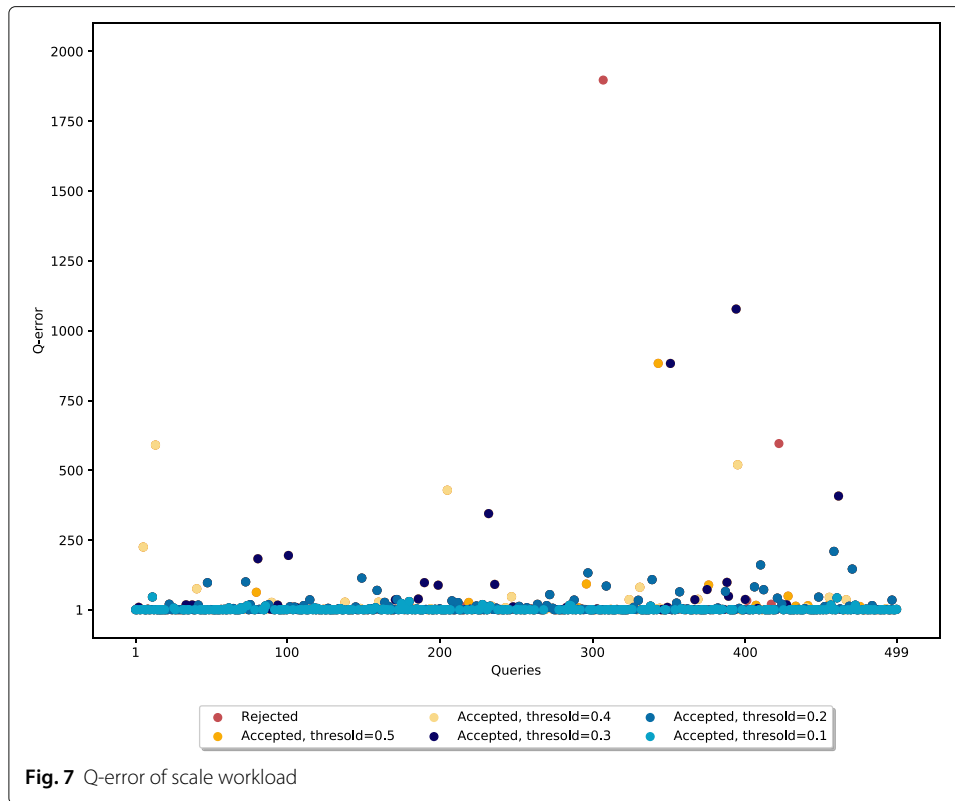
When we analyze Table 5, we conclude again that this type of queries can be avoided by applying the uncertainty management of our model. For example, in the most restrictive threshold (0.1), we have that the number of accepted queries is 246 (almost half of the workload queries) whose 99th percentile of the Q-error is at most 27.2, being a very small estimation error. In other words, outliers that make Robust Cardinality's mean and maximum sensitive are avoided in managing threshold, even when a threshold value of 0.5 is used. Note that at this threshold value, six queries were still rejected.

To demonstrate the impact of uncertainty management on Robust Cardinality's estimates, let us look at the distribution of accepted queries for each threshold in Fig. 7. Again, we had to remove queries outliers that showed a very large Q-error so that the scale of the plot could be smaller. Thus, we limit the queries that presented a Q-error up to 2000. See in the figure that the light blue dots represent the queries that were accepted with a threshold of 0.1, the most restrictive one. Therefore, it is clear that this level of threshold limits the Q-error to be around the ideal value 1. That is, the cardinality estimates of these queries are quite accurate in our technique, which guarantees that the queries accepted at that threshold value will be estimated with good accuracy. Also note that a threshold of 0.2 still has a low level of Q-error. Finally, it is clear that as the threshold is relaxed, the Q-error increase in accepted queries occurs.

Lastly, we now analyze the results obtained in the execution of the JOB workload. In the first part of the analysis, we observe the values presented in Table 6. While PostgreSQL showed a better mean, 99th percentile and maximum value, in the other metrics the MSCN won over the other two techniques. In fact, the MSCN, especially in this workload, presented the best result, since it had a higher mean than PostgreSQL due to its maximum value being much higher than that, in addition to having a greater sensitivity

Table 5 Scale queries accepted and rejected and their Q-errors for each threshold value

Threshold	0.1		0.2		0.3		0.4		0.5	
	Acc	Rej	Acc	Rej	Acc	Rej	Acc	Rej	Acc	Rej
# of queries	246	253	371	128	437	62	475	24	493	6
99% Q-error	27.2	976.4	119.9	1676.0	296.4	1278.6	452.7	1664.1	525.7	1832.3



for queries the 99th percentile. But in all other metrics, the MSCN got better values. Note that the MSCN produced better estimates up to the 95th percentile, but from then on the accuracy started to deteriorate. That is, in general, the MSCN presented the smallest cardinality estimation errors.

Although our technique showed Q-error values greater than the MSCN, the values up to the median were not so far apart as our median was approximately 12, less than twice as high as that of the MSCN. In the rest, our errors were much higher, even though our maximum error was 6000, while that of the MSCN was approximately 4500. However, these maximum errors are much higher than the error obtained by PostgreSQL, even though this technique has worse results than the MSCN overall. So, at first glance, it can be concluded that our technique behaved worse than the other two in this workload. Nevertheless, we now move on to the second part of our analysis.

In the second part of the analysis, we include the management of uncertainty, this is where our technique shows its strength in relation to the others. In Table 7, we present the results of the queries that were accepted or rejected according to an acceptance threshold. In particular, we note that 16 queries were accepted with a threshold equal to 0.1, with

Table 6 Aggregated Q-error of cardinality estimates of JOB workload by each model (lower is better)

Technique	Mean	Median	Percentile			Maximum
			90	95	99	
PostgreSQL	34.411	9.841	102.475	127.213	255.637	280.122
MSCN	86.983	6.622	85.548	124.868	1,572.327	4,506.700
Robust Cardinality	554.055	12.368	1,859.547	2,874.295	4,712.486	6,004.769

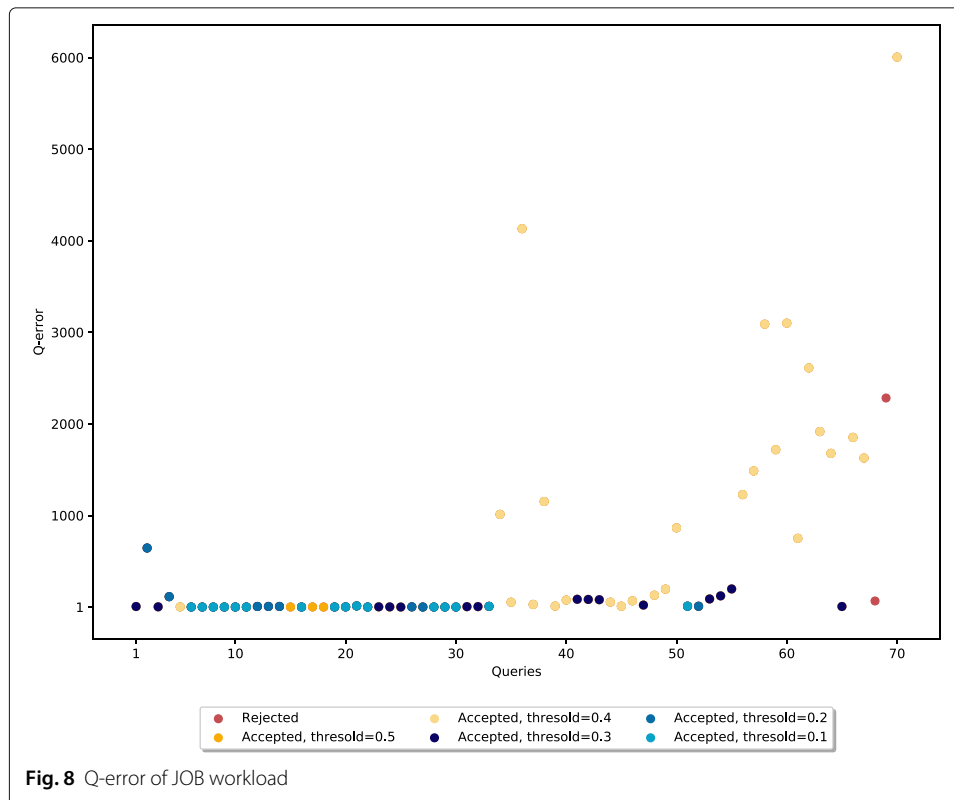
Table 7 JOB queries accepted and rejected and their Q-errors for each threshold value

Threshold	0.1		0.2		0.3		0.4		0.5	
# of queries	Acc	Rej	Acc	Rej	Acc	Rej	Acc	Rej	Acc	Rej
99% Q-error	12.7	5012.2	523.3	5162.0	475.6	5442.9	4806.1	2194.4	4749.9	2260.8

99th percentile of the Q-error of these queries being 12.7, which is well below the values presented in the previous table of all three techniques, which shows that at this level of low uncertainty it is possible for our technique to present very accurate estimates. Also, note that the threshold with a value of 0.2 has a good value for the 99th percentile of the Q-error, in addition to allowing a greater number of queries to be accepted, in this case 24 queries. Therefore, even though our model presented less accurate estimates in general for this workload, as seen earlier, when considering uncertainty management, it is possible to obtain good cardinality estimates for this workload as well.

To finish our analysis, Fig. 8 shows how the queries were distributed according to the level of accepted uncertainty defined through the threshold. Please note that the light blue dots are very close to the value 1, which shows that these queries have good cardinality estimates. Similarly, points whose threshold is equal to 0.2 also have good estimates.

Finally, an important point to note is the fact that the outlier query that presented a Q-error of approximately 6000 was accepted with a threshold of 0.5, but two other queries, represented by red dots, with much smaller Q-errors were not accepted with a threshold of at most 0.5. This situation is possible to occur considering that, as stated in the “Uncertainty” section, the DBA must choose what level of confidence he wants for the



uncertainty of the queries to be evaluated. In our case, the confidence level chosen was 90%, which means that in 90% of the cases our uncertainty assessment will be carried out correctly.

In summary, while MSCN may constantly be off by an amount of error, Robust Cardinality's error is lower, but when an estimation error does happen, it may be very high. Nevertheless, this is mitigated by the configurable confidence level. PostgreSQL's histogram results were not commented with details because they happen to have the worst overall than the two machine learning approaches. It confirms that the Robust Cardinality model is efficient in its predictions.

Query performance

After evaluating the quality of the cardinality estimates generated by our technique, we will now be interested in making an analysis of the impact that the gain brought by Robust Cardinality in the estimation process can have on the execution of the queries itself, that is, how the execution time of the queries will be impacted. For the overhead incurred by adding Robust Cardinality over PostgreSQL, 5000 synthetic queries were executed over a clean install of PostgreSQL and one with Robust Cardinality. Model evolution was not considered in this experiment. First, let us look at the accumulated execution times, in minutes, that are shown in Table 8 in order to assess the impact during the execution of the queries. For that, we present the values for when 1250, 2500, 3750, and 5000 queries have already been performed.

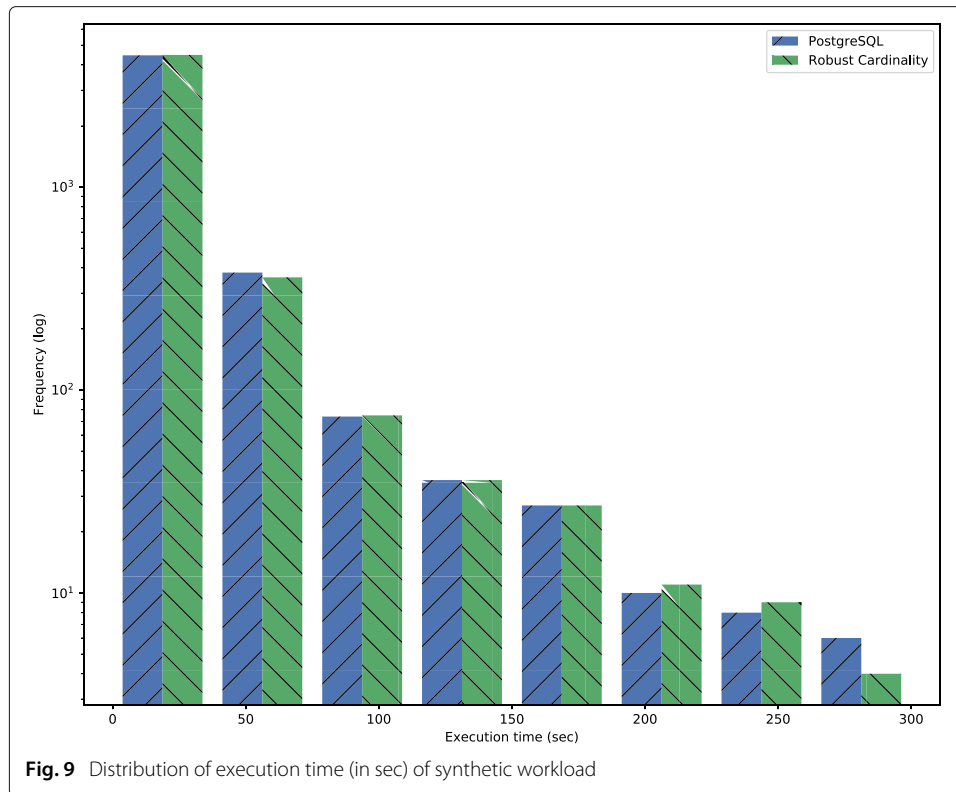
This table shows that, even with the overhead imposed by Robust Cardinality, execution times are lower than with stock PostgreSQL. To detail this, Fig. 9 shows the distribution of queries according to their execution time (in s) in both the clean PostgreSQL and Robust Cardinality implemented in it.

Figure 9 compares distribution of query execution time using PostgreSQL histogram technique and Robust Cardinality. It is hard to appoint any impact of the enhancement of cardinality estimation using Robust Cardinality in the execution time of queries that have a shorter execution time than 200 s. Nevertheless, it is possible to observe that queries that have execution time greater than this value were performed considerably better on Robust Cardinality. In particular, note that the frequency of queries close to three hundred seconds has been reduced by approximately half when performed using our estimation technique. Hence, we conclude that our technique has a greater impact on queries that have a long execution time.

Another way to visualize that the improvement brought by Robust Cardinality is in those queries with a long execution time, observe Fig. 10 which presents the following values: the mean, standard deviation and 90th percentile of the execution time of the queries using both solutions. PostgreSQL histogram and Robust Cardinality show similar mean and standard deviation metrics, but Robust Cardinality presents lower 90% percentile. It

Table 8 Accumulated execution time in minutes for 1250, 2500, 3750, and 5000 queries over PostgreSQL and Robust Cardinality

Technique	≤ 1250	≤ 2500	≤ 3750	≤ 5000
PostgreSQL	285.149	612.498	906.343	1,210.445
Robust Cardinality	277.094	598.768	886.042	1,182.964
Difference	8.055	13.730	20.301	27.481



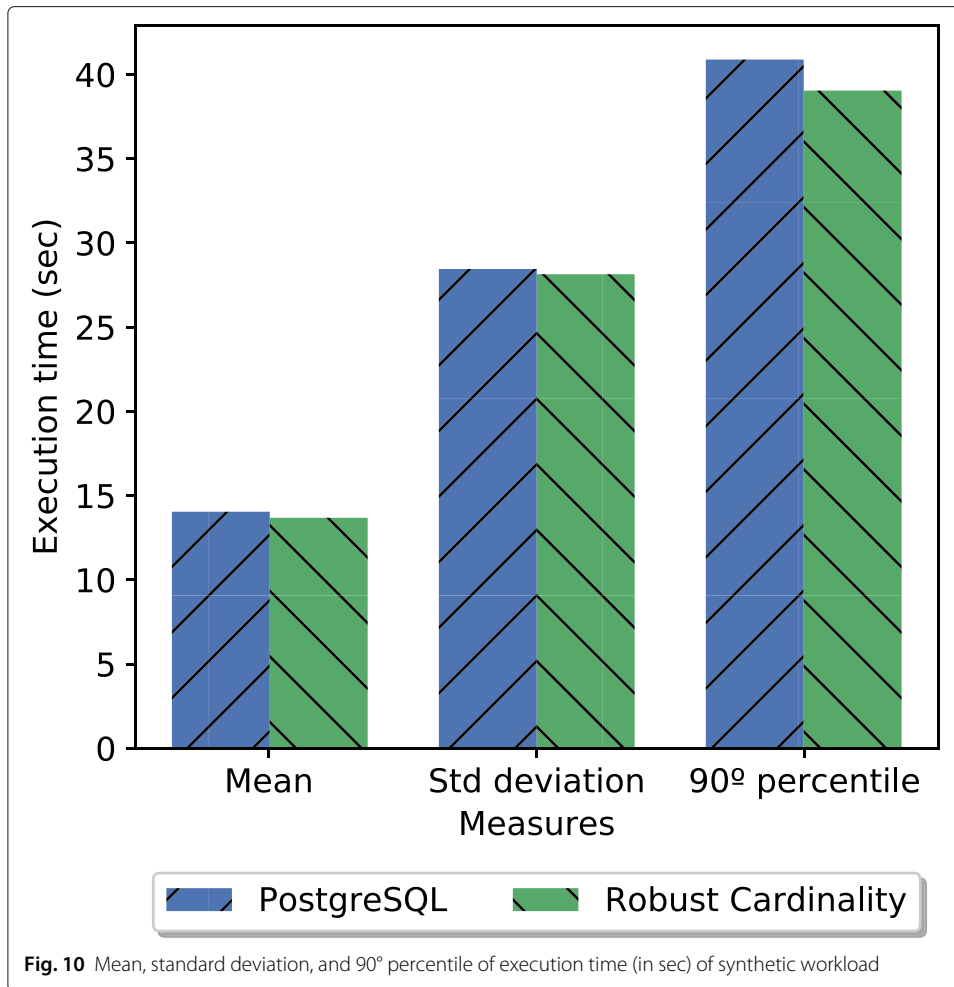
implies that using Robust Cardinality in query processing helps to choose faster query plans for slower queries. To demonstrate this, the selected execution plans for a given query are shown in Fig. 11, with left being the plan selected using histogram estimates and right the plan selected using Robust Cardinality estimates.

The intermediate tables' size is much smaller in the Robust Cardinality plan. The highest join in the tree is joining 42,965,075 tuples with 104,281 tuples on intermediate tables in the plan generated by PostgreSQL, while the table sizes in Robust Cardinality are 104281 and 17. By using cardinalities closer to the real value, a plan using index scan and a nested loop join was selected over a plan that did hash joins and a sequential scan, because the original estimates misled the optimizer to believe that the hash join's cost would be enough to mitigate the sequential scan high cost.

Model evolution

To observe how the model evolves, we performed an experiment in which the model is gradually trained and the Q-error metric is evaluated. This experiment shows how the model accumulates experience over time, not wasting previous knowledge of the workload. Figure 12 shows how the Q-error decreases given a few rounds of training.

The Q-error highly decreases from the first iteration. The fact that no increases happen afterward, together with the rapid decrease in the early iterations shows that the model retains the previous knowledge and enhances it with newly observed samples, achieving a behavior of evolution. Moreover, as it is possible to manage uncertainty, then the model can now be used to generate cardinality estimates throughout the training process for

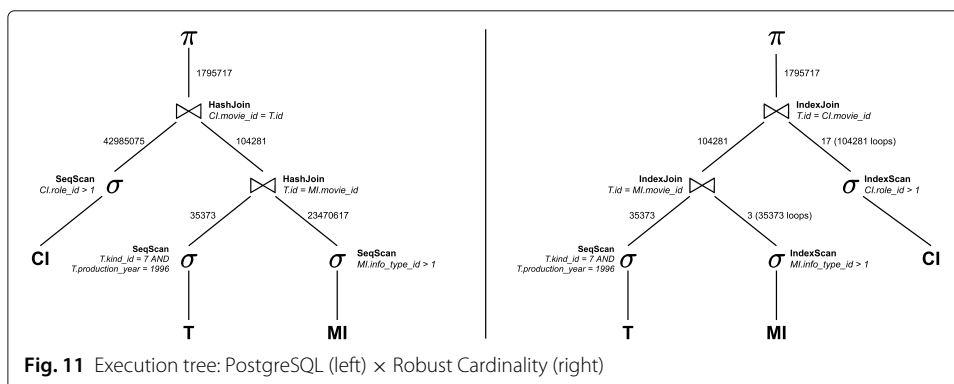


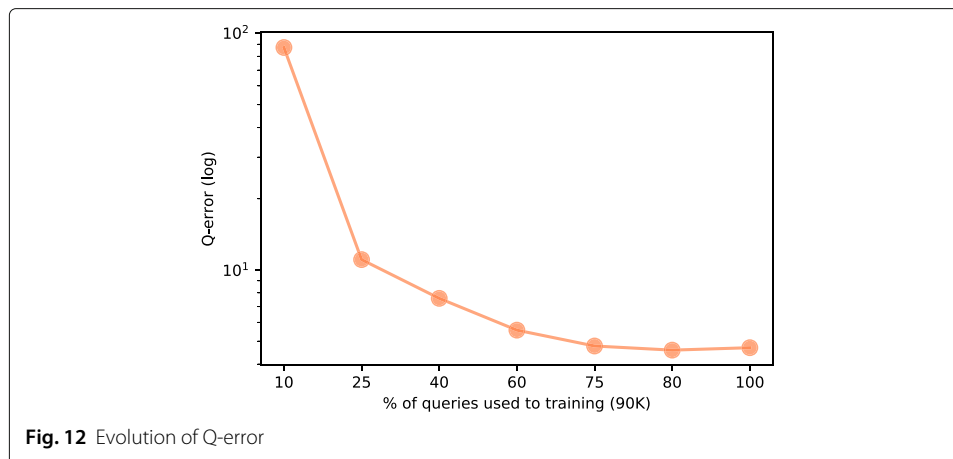
those queries that have low uncertainty at that time. Thus, the model is robust and can learn and evolve, rejecting estimates in which it is not confident upon.

General discussion

Takeaways

Our results suggest that the incorporation of machine learning models can bring benefits to the processing of queries in DBMSs. First, this type of models is able to generate good





cardinality estimates. Second, the uncertainty about the estimates is a very useful metric, and its management may help the DBMS to avoid using poor cardinality estimates in query processing. In our case, it guarantees that the DBMS will rarely use estimates worse than the standard method. Third, it is possible to successfully merge a standard strategy, proved by time and fruit of years of study and development, with a new solution with great potential that is based on machine learning. Finally, we demonstrated with Robust Cardinality that it is possible to build a modular solution that requires little preparation of the DBMS.

Limitations

As we see in the experimental evaluation presented above, although Robust Cardinality may come with significant improvements, there is still much work to improve it. There are some limitations to this approach that must be highlighted. This approach does not handle changes in the tables' schema, and it is easy to understand why. The representation of queries in vectors, which is essential to feed the GBDT model, is bound to the schema. For example, the first section of the vector representing the query's tables depends on the number of tables. Besides that, the encoding does not consider many possible operations in SQL, likewise ordering, aligned queries, partial joins, group by, and distinct.

Furthermore, we could not test this approach over an extended period of time. The successive increment of new decision trees in the GBDT model may generate unpredictable results. In the scope of this work, we do not implement a forgetting mechanism.

Conclusion and future work

This paper presented Robust Cardinality, a plug-and-play cardinality estimation technique. We have shown that our approach is efficient to estimate query cardinality in the optimization phase query processing. Moreover, it builds machine learning models with very low overhead on the query execution time. Robust Cardinality allows for model evolution accumulating previous knowledge to build new models. Our approach also manages inaccurate predictions by supporting a confidence interval on query cardinality predictions. Whenever cardinality estimation is outside this interval, a conservative decision is made for the DBMS to execute the best query plan found by its regular optimizer.

We have conducted experiments using PostgreSQL to support our contributions. Experimental results have shown that our approach can generate 4x lower error in query cardinality estimates and; therefore, it achieves better query runtime performance in approximately 3% in the overall execution time. Hence, Robust Cardinality can be implemented in real-world systems with little overhead and can learn the data's intrinsic characteristics through the queries, as our results show.

In the future, other types of queries can be investigated, such as grouping, ordering, and nested queries. So far, no works have considered these types of operators. Also, the technique only allows numeric predicates. Therefore, a new encoding technique can be proposed to use more complex predicates, such as strings and operators. Finally, an investigation into the possibility of performing a deeper integration of machine learning techniques with the internal DBMS query processing architecture is an exciting research subject.

Abbreviations

DBMS: Database management system. GBDT: Gradient boosting decision tree. IMDb: Internet Movie Database. MSCN: Multi-set convolutional network. LightGBM: Light gradient boosting machine. LSB: Laboratory of systems and databases. UFC: Federal University of Ceará

Acknowledgements

The authors would like to thank Laboratory of Systems and Databases (LSB), Federal University of Ceará (UFC), and the funding agency.

Authors' contributions

This work was mainly done by FDBSP while he was a graduate student supervised by JCM. Both are the main authors of this research. ICA and FLFP helped them in implementation of Robust Cardinality. PRPA contributed on analyzing the experimental results and writing the manuscript. All authors read and approved the final manuscript.

Funding

This work was partly funded by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior — Brasil (CAPES) — grant 1782887.

Availability of data and materials

IMDB used during the current study can be obtained in the following link: <https://imdbpy.readthedocs.io/en/latest/>. The queries used and/or analyzed during the current study are available from the corresponding author on reasonable request.

Declarations

Competing interests

The authors declare that they have no competing interests.

Received: 26 August 2020 Accepted: 30 July 2021

Published online: 01 September 2021

References

1. Kooi RP (1980) The optimization of queries in relational databases. PHD Thesis. Case Western Reserve University
2. Leis V, Gubichev A, Mirchev A, Boncz PA, Kemper A, Neumann T (2015) How good are query optimizers, really? Proc VLDB Endowment 9(3):204–215
3. Leis V, Radke B, Gubichev A, Mirchev A, Boncz PA, Kemper A, Neumann T (2018) Query optimization through the looking glass, and what we found running the join order benchmark. VLDB J 27(5):643–668
4. Ioannidis YE, Christodoulakis S (1991) On the propagation of errors in the size of join results. In: Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data. pp 268–277. <https://doi.org/10.1145/115790.115835>
5. Moerkotte G, Neumann T, Steidl G (2009) Preventing bad plans by bounding the impact of cardinality estimation errors. Proc VLDB Endowment 2(1):982–993
6. Aken DV, Pavlo A, Gordon GJ, Zhang B (2017) Automatic database management system tuning through large-scale machine learning. In: Proceedings of the 2017 ACM International Conference on Management of Data. pp 1009–1024
7. Pavlo A, Angulo G, Arulraj J, Lin H, Lin J, Ma L, Menon P, Mowry TC, Perron M, Quah I, Santurkar S, Tomasic A, Toor S, Aken DV, Wang Z, Wu Y, Xian R, Zhang T (2017) Self-driving database management systems. In: 8th Biennial Conference on Innovative Data Systems Research
8. Lima MIV, de Farias VAE, Praciano FDBS, Machado JC (2018) Workload-aware parameter selection and performance prediction for in-memory databases. In: SBBD. pp 169–180

9. Graefe G, Kuno HA (2010) Self-selecting, self-tuning, incrementally optimized indexes. In: Manolescu I, Spaccapietra S, Teubner J, Kitsuregawa M, Léger A, Naumann F, Ailamaki A, Özcan F (eds). Proceedings of the 13th International Conference on Extending Database Technology Vol. 426. pp 371–381
10. Teixeira EM, Amora PRP, Machado JC (2018) MetisIDX - from adaptive to predictive data indexing. In: EDBT. pp 485–488
11. Kraska T, Beutel A, Chi EH, Dean J, Polyzotis N (2018) The case for learned index structures. In: Proceedings of the 2018 International Conference on Management of Data. pp 489–504
12. Krishnan S, Yang Z, Goldberg K, Hellerstein JM, Stoica I (2018) Learning to optimize join queries with deep reinforcement learning. CoRR. <https://doi.org/abs/1808.03196>
13. Marcus R, Papaemmanouil O (2019) Towards a hands-free query optimizer through deep learning. In: 9th Biennial Conference on Innovative Data Systems Research
14. Ortiz J, Balazinska M, Gehrke J, Keerthi SS (2018) Learning state representations for query optimization with deep reinforcement learning
15. Harmouch H, Naumann F (2017) Cardinality estimation: An experimental survey. Proc VLDB Endowment 11(4):499–512. <https://doi.org/10.1145/3186728.3164145>
16. Yin S, Hameurlain A, Morvan F (2015) Robust query optimization methods with respect to estimation errors: A survey. ACM Sigmod Rec 44(3):25–36. <https://doi.org/10.1145/2854006.2854012>
17. Lipton RJ, Naughton JF, Schneider DA (1990) Practical selectivity estimation through adaptive sampling. In: Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data. pp 1–11
18. Stillger M, Lohman GM, Markl V, Kandil M (2001) LEO - db2's learning optimizer. In: Proceedings of 27th International Conference on Very Large Data Bases. pp 19–28
19. Chaudhuri S, Narasayya VR, Ramamurthy R (2008) A pay-as-you-go framework for query execution feedback. Proc VLDB Endowment 1(1):1141–1152
20. Liu H, Xu M, Yu Z, Corvinelli V, Zuzarte C (2015) Cardinality estimation using neural networks. In: Proceedings of 25th Annual International Conference on Computer Science and Software Engineering. pp 53–59
21. Zhou X, Chai C, Li G, SUN J (2020) Database meets artificial intelligence: a survey. IEEE Transactions on Knowledge and Data Engineering:1–1. <https://doi.org/10.1109/TKDE.2020.2994641>
22. Dutt A, Wang C, Nazi A, Kandula S, Narasayya VR, Chaudhuri S (2019) Selectivity estimation for range predicates using lightweight models. Proc VLDB Endowment 12(9):1044–1057
23. Kipf A, Kipf T, Radke B, Leis V, Boncz PA, Kemper A (2019) Learned cardinalities: estimating correlated joins with deep learning. In: 9th Biennial Conference on Innovative Data Systems Research
24. Zaheer M, Kottur S, Ravanbakhsh S, Poczos B, Salakhutdinov RR, Smola AJ (2017) Deep sets. In: Advances in neural information processing systems. pp 3391–3401
25. Woltmann L, Hartmann C, Thiele M, Habich D, Lehner W (2019) Cardinality estimation with local deep learning models. In: Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management. pp 1–8
26. Negi P, Marcus R, Mao H, Tatbul N, Kraska T, Alizadeh M (2020) Cost-guided cardinality estimation: focus where it matters. In: 2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW). pp 154–157
27. Breiman L, Friedman J, Olshen R, Stone C (1984) Classification and regression trees, wadsworth statistics. Probability Series, Belmont, California: Wadsworth
28. Friedman JH (2001) Greedy function approximation: a gradient boosting machine. Ann Stat:1189–1232
29. Sollich P, Krogh A (1996) Learning with ensembles: how overfitting can be useful. In: Advances in Neural Information Processing Systems. pp 190–196
30. Marqués AI, García V, Sánchez JS (2012) Exploring the behaviour of base classifiers in credit scoring ensembles. Expert Syst Appl 39(11):10244–10250
31. Rogozhnikov A (2016) Gradient boosting explained [demonstration]. http://arogozhnikov.github.io/2016/06/24/gradient_boosting_explained.html. Access in 01-15-2020
32. Jansen S (2018) Hands-On Machine Learning for Algorithmic Trading: Design and implement investment strategies based on smart algorithms that learn from data using Python. Packt Publishing Ltd
33. Hashem S, Schmeiser B (1995) Improving model accuracy using optimal linear combinations of trained neural networks. IEEE Trans Neural Netw 6(3):792–794
34. Breiman L (1996) Bagging predictors. Mach Learn 24(2):123–140
35. Schapire RE, Singer Y (1999) Improved boosting algorithms using confidence-rated predictions. Mach Learn 37(3):297–336
36. Bauer E, Kohavi R (1999) An empirical comparison of voting classification algorithms: bagging, boosting, and variants. Mach Learn 36(1-2):105–139
37. Kazienko P, Lughofer E, Trawinski B (2013) Hybrid and ensemble methods in machine learning. J Univ Comput Sci 19(4):457–461
38. Meinshausen N (2006) Quantile regression forests. J Mach Learn Res 7:983–999
39. Freund Y, Schapire RE, et al (1996) Experiments with a new boosting algorithm. In: Icml Vol. 96. pp 148–156. Citeseer
40. Jones E, Oliphant T, Peterson P, et al (2001) SciPy: open source scientific tools for Python. <http://www.scipy.org/>
41. Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E (2011) Scikit-learn: machine learning in Python. J Mach Learn Res 12:2825–2830
42. Ke G, Meng Q, Finley T, Wang T, Chen W, Ma W, Ye Q, Liu T-Y (2017) Lightgbm: a highly efficient gradient boosting decision tree. Adv Neural Inform Process Syst 30:3146–3154
43. Leis V, Radke B, Gubichev A, Kemper A, Neumann T (2017) Cardinality estimation done right: index-based join sampling. In: 8th Biennial Conference on Innovative Data Systems Research

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.