PostCENN: PostgreSQL with Machine Learning Models for Cardinality Estimation

Lucas Woltmann
Dominik Olwig
Claudio Hartmann
Dirk Habich
Wolfgang Lehner
Database Systems Group, TU Dresden
Dresden, Germany
firstname.lastname@tu-dresden.de

ABSTRACT

In this demo, we present *PostCENN*, an enhanced PostgreSQL database system with an end-to-end integration of machine learning (ML) models for cardinality estimation. In general, cardinality estimation is a topic with a long history in the database community. While traditional models like histograms are extensively used, recent works mainly focus on developing new approaches using ML models. However, traditional as well as ML models have their own advantages and disadvantages. With *PostCENN*, we aim to combine both to maximize their potentials for cardinality estimation by introducing ML models as a novel means to increase the accuracy of the cardinality estimation for certain parts of the database schema. To achieve this, we integrate ML models as first class citizen in PostgreSQL with a well-defined end-to-end life cycle. This life cycle consists of creating ML models for different sub-parts of the database schema, triggering the training, using ML models within the query optimizer in a transparent way, and deleting ML models.

PVLDB Reference Format:
Lucas Woltmann, Dominik Olwig, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. *PostCENN*: PostgreSQL with Machine Learning Models for Cardinality Estimation. PVLDB, 14(12): 2715 - 2718, 2021.
doi:10.14778/3476311.3476327

1 INTRODUCTION

With the ever increasing amount of data, efficient query processing in database management systems is still a major challenge. Thus, optimizing query processing is an active research topic. Besides many other aspects, the determination of an optimal query execution plan is particularly important. To find the optimal plan, query optimizers typically rely on size estimations of base tables as well as intermediate results. The quality of these estimates has a crucial impact on the plan optimality and Leis et al. have shown that traditional estimators like histograms cannot provide sufficiently accurate estimates to make sure the optimal plan can be found [3]. To overcome this issue, current research successfully applies Machine Learning (ML) techniques. Especially, Neural Networks (NNs) are capable to provide high quality cardinality estimates [2, 4, 8].

However, these works only show improvements by evaluating their accuracy with the deviation of the estimates from the actual cardinalities in an isolated way. Therefore, they do not provide insights into the actual query performance gains. Some approaches use their estimators as external tools, but also do not fully integrate them into a database management system [1, 5]. This partial integration approach adds a *not negligible* overhead to the runtime of a query since the call to the external estimator is usually slow compared to the highly optimized query execution engines.

To overcome these shortcomings, we present *PostCENN*, an enhanced PostgreSQL database system with an end-to-end integration of NNs for cardinality estimation in this demo. Moreover, this enables users to dynamically switch between NNs and traditional estimators to find whatever suits them best as shown in [6]. Our integration is mainly driven by the core idea to introduce NNs as first class citizen as a means to optimize the accuracy of the cardinality estimation. Thus, our approach has many similarities to index structures which can be flexibly created and utilized to speed up data access. Our end-to-end integration covers all life cycle aspects: from creating NNs over the database-optimized training of NNs to the utilization of NNs during the query optimization phase. As conceptual foundation, we use our local model approach for cardinality estimation [8]. Local models are NNs covering only a certain sub-part of the database schema as their model context enabling flexible use. A main drawback of these NNs is the high construction cost by executing a lot of example queries within the model context. To tackle this problem, we developed a novel approach based on *pre-aggregating* the base data of the model context [7] and executing the example queries on this pre-aggregated data. To efficiently realize this *pre-aggregation*, we create a *grouping set* within the database for storing and computing aggregated information. Finally, we enhance the query optimizer to request cardinality estimations from these NNs if an appropriate NN is available.

Based on this, the main contributions of this demo are:

- We introduce the implementation for our *PostCENN* system by describing all aspects for an end-to-end integration of NNs into PostgreSQL in Section 2.
- In our demonstration, we give a detailed explanation of all *PostCENN* concepts. In particular, our demo —as described in

Proceedings of the VLDB Endowment, Vol. 14, No. 12 ISSN 2150-8097.
doi:10.14778/3476311.3476327

Section 3— shows (i) how to flexibly create NNs for specific sub-parts of the database schema, (ii) how our improved NN construction works and performs, and (iii) how single queries as well as benchmarks improve with better estimates.

Finally, we briefly summarize our demo paper in Section 4.

2 SYSTEM DESCRIPTION

Cardinality estimation is a topic with a long history in database research and traditional approaches are histograms, sampling, and sketching [3, 6]. However, these approaches are typically based on statistical models with simplifying assumptions like uniformity and independence [3]. This leads to erroneous estimates in situations with complex data dependencies and correlations between attributes. As shown in [2, 3, 8], neural networks (NNs) can be used to provide high accuracy estimates in such situations. However, main shortcomings of NNs are the high construction costs rooted in their supervised learning approach. So, traditional as well as NN approaches have their own advantages and disadvantages. Thus, as described in [6], our overall goal is to use the best of both worlds and bring traditional and NN models together to maximize their potential for cardinality estimation.

To achieve this goal and to foster further research in that direction, we developed *PostCENN* integrating our previous research results [6–8] into the open-source database system PostgreSQL in an end-to-end fashion. PostgreSQL usually uses a traditional histogram-based approach for cardinality estimation and this established approach remains the standard in *PostCENN*. To flexibly improve the estimation quality for sub-parts of the database schema, we integrated NNs as first class citizen using our local model approach [8]. For our integration, we borrow a lot from index structures. Just like indexes, NNs are additional data structures covering only a specific sub-part of the database schema. Hence, we also mirror the life cycle of index structures to our NN models.

2.1 Local Model Foundation

As presented in [8], our local model approach improves cardinality estimation for a sub-part of the database schema [8]. To do so, a local model is defined for a specific *context* and the corresponding NN model can only be used for that context. A context can be any base table or any join of tables. For each context, a separate NN model has to be trained with example queries. After training, the NN can be used as a cardinality estimator through its forward pass application. The advantage of our local models compared to other ML-based approaches is their light structure, fast training, and easy deployment. They are also capable of producing cardinality estimates with high quality, i.e. low estimation errors (q-error).

2.2 Local Model Life Cycle

To have our local NN models as first class citizen in *PostCENN*, we defined a unified life cycle for each local NN model as illustrated in Figure 1. This life cycle consists of the following stages: (i) creating a local context, (ii) triggering the training and therefore creating a model, (iii) using local models within the query optimizer in a transparent way, and (iv) deleting a context or model. In the following sections, we describe each stage individually.

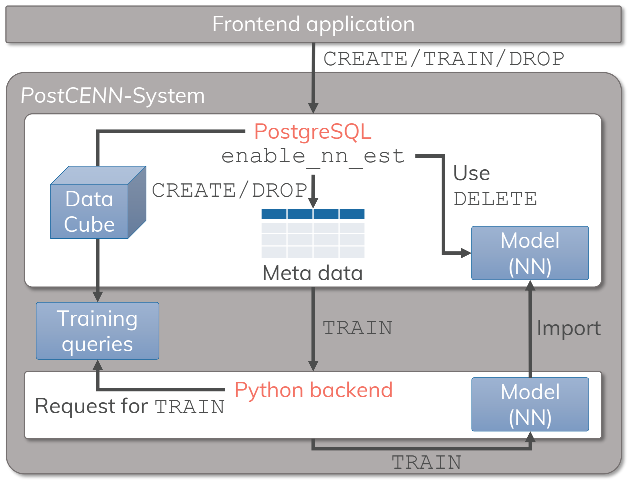


Figure 1: *PostCENN* system architecture.

Creating a Local Context. This stage is equivalent to defining a collection of tables, their columns, and their join attributes to identify a context. A context is stored as a reference for every model in it. The user can create contexts over all tables in a database schema. Furthermore, the user can limit the context to a specific set of columns. This is important if the user only wants to examine parts of a context or if the context's tables are very large. All information about the context, like total tuple count or column ranges, are stored in meta information tables within the database. Just like an index, we enable the user to create a model context by calling `CREATE CONTEXT`:

```
CREATE CONTEXT context_name
    ON table_name(column1, column2, ...)
    [, table_name(column1, column2,...)]
    [USING join_ids];
```

Using this statement, users only create a local context with a unique `context_name` within the database. The corresponding NN model is not automatically trained due to the required construction time.

Training a Local NN Model. The training process is the most time-consuming part of the usage of NNs for cardinality estimation. Thus, this training should be done asynchronously and only if triggered by the user with an explicit command. For creating a trained local model, the users select a context for which they want to train an NN. For this training, the user has to specify NN-specific parameters such as width, depth, and number of example queries as described in [8]. Then, an NN is configured and trained with these parameters using the following statement:

```
TRAIN MODEL model_name
    ON context_name
    USING (model_width, model_depth,
        number_of_example_queries);
```

This `TRAIN MODEL` statement creates a static NN model on the specified context. Static means, the NN model is trained once and kept as it is until a new training is triggered manually. The maintenance of these local models is an interesting future research topic and currently out-of-scope of this demo.

To efficiently train our NNs, we tightly coupled the underlying PostgreSQL with a Python backend containing Tensorflow in *PostCENN* as illustrated in Figure 1. Therefore, the training of the NN models is outsourced to Tensorflow, whereas our Python backend is responsible (i) to query the necessary example data from the database, (i) to conduct the necessary vectorization of the example queries, and (iii) to manage the training on GPUs.

To train an NN, or more specifically: to learn a supervised ML model, many pairs consisting of `(example_query, output_cardinality)` over the model context are required as input. To determine the correct `output-cardinalities`, the example queries are rewritten with a count aggregate and executed individually. The example queries are restricted to the model context, such that they mainly differ in their predicates. To optimize the execution of all example queries for a context, we develop a solution using a *predicate-independent pre-aggregation* of the context data and execute the example queries over this pre-aggregated data. Consequently, the set of similar example queries has to read and process less data because the pre-aggregation is a compact representation of the context base data. To realize this pre-aggregation, we use the *grouping set* construct, a special type of data cube, for storing and computing aggregation information which is available in PostgreSQL. However, this pre-aggregation is only beneficial if the execution of the example queries on the grouping set plus the time for creating the grouping set is faster than the execution of the example queries over the base data [7]. Thus, we internally evaluate the benefit and the pre-aggregation is only conducted if it is advantageous. If this is not the case, the example queries are executed on the context base data.

After finishing the training, the meta data for the corresponding context within the database is updated with an additional reference to the trained model. The NN model is now available and can be used. A context can have more than one trained model. This is particularly important to the users if they want to train models with different parameters for width, depth, or number of example queries and compare the different resulting NNs to each other.

Usage of Local Models. The main application area of our NN models is within the query optimizer. Here, the query optimizer usually requests an estimated cardinality for a specific (sub-)query from an NN model. This procedure is called *forward pass*. The forward pass is pretty much encapsulated in the query optimizer, so we decided to directly integrate tensorflow's NN management capabilities into PostgreSQL using the *Tensorflow C API*[1] to achieve the best performance. Moreover, Tensorflow models are exported from the Python code in a cross-platform format. We use this feature to import the model binary into the running PostgreSQL instance, such that the forward pass is always executed within PostgreSQL without touching any external tool.

To control the usage of NNs, we additionally introduced a parameter called `enable_nn_est`. It can be toggled via:

```
SET nn_enable_est TO [on/off];
```

[1]https://www.tensorflow.org/install/lang_c

If this parameter is set to `on` all available NNs will be used transparently if their context is queried. This also applies for queries where the context is only a part of a larger query construct or for subqueries. If `enable_nn_est` is set to `off`, the standard PostgreSQL estimator is used in all cases. This design choice arises from the general property of local models that they are only initialized where it appears beneficial to the user [6, 8]. Therefore, an individual decision for every single context is not required. However, we give the users the control over which trained model they want to use for a specific query if the query accesses the matching context. This again helps to compare the potential of models with different parameter configurations to each other.

Deleting a Local Model. Deleting a model or a context is rather simple compared to the previous steps. To trigger the removal, the user has two options:

```
DROP CONTEXT context_name;
DROP MODEL model_name;
```

If a user triggers the removal of a context by using the `DROP CONTEXT` statement, *PostCENN* deletes all models associated to the context as well as the context. If the user executes the `DROP MODEL`, only the corresponding model will be removed from the system.

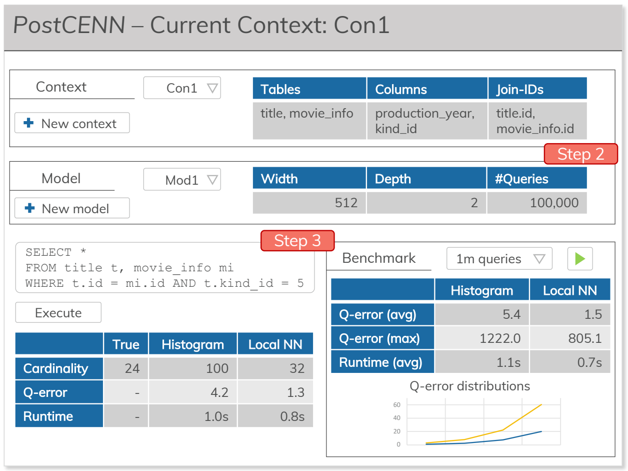
3 DEMONSTRATION DETAILS

The overall aim of our VLDB demo is to present *PostCENN* and its end-to-end integration concept of NNs to improve the cardinality estimation for sub-parts of the database schema. For this, we address four aspects in our demo: (i) we introduce our *PostCENN* design including all necessary concepts, (ii) we walk through the life cycle of local models, (iii) we show the benefits of our best of both worlds approach for query optimization, and (iv) we present open issues for further research. Most importantly, we would like to give demo visitors a comprehensive understanding of our entire *PostCENN* approach. To achieve this, we provide an interactive graphical user interface as a front-end for *PostCENN*. Throughout the demo, we are using the IMDB[2] data set and corresponding queries on it. Based on this, a demo visitor experiences four steps.

Step 1 - Concept Explanation. At the beginning, every demo visitor gets an explanation of all *PostCENN* concepts. This includes the workings of local models, their construction and training, and their use as a cardinality estimator in the query optimization phase. Additionally, we explain our *predicate-independent pre-aggregration* optimization for the training phase. For this explanation, we prepare some slides as well as a poster to interactively present our concepts. All these concepts can then be tested in our *PostCENN* prototype.

Step 2 - Life Cycle Walk-Through. Next, we guide demo visitors through the life cycle of local models to mainly demonstrate our integration of the local model management into SQL. Here, visitors have the opportunity to create, train, and delete predefined local models in our front-end as shown in Figure 2. Additionally, they can setup their own contexts within the IMDB database and train models on a limited scale. Then, our front-end reports progress and results of the integration. Moreover, visitors can retrace our optimization approach for the model training. We also use this step, to

[2]ftp://ftp.fu-berlin.de/pub/misc/movies/database/frozendata/


PostCENN – Current Context: Con1

Context: Con1

+ New context

Tables	Columns	Join-IDs
title, movie_info	production_year, kind_id	title.id, movie_info.id

Step 2

Model: Mod1

+ New model

Width	Depth	#Queries
512	2	100,000

Step 3

```
SELECT *
FROM title t, movie_info mi
WHERE t.id = mi.id AND t.kind_id = 5
```

Execute

	True	Histogram	Local NN
Cardinality	24	100	32
Q-error	-	4.2	1.3
Runtime	-	1.0s	0.8s

Benchmark: 1m queries

	Histogram	Local NN
Q-error (avg)	5.4	1.5
Q-error (max)	1222.0	805.1
Runtime (avg)	1.1s	0.7s

Figure 2: The interactive graphical user interface of our demonstration (Step 2 and 3).

explain the interaction of the main *PostCENN* system components PostgreSQL and Python backend in detail. Furthermore, we open the evaluation of the training process to the user by revealing the optimization of the example query generation as presented in [7]. Furthermore, all hyperparameters of the specific local NNs [8] are adjustable to improve the training process quality. So, different models, with and without our optimization, can be examined regarding their q-errors and resulting query runtimes for both single queries and workloads.

Step 3 - Benefit for Query Optimization. In this step, we would like to convey demo visitors an understanding on the benefit of our approach for query optimization. Visitors can directly compare cardinality estimates of histograms and NNs for a single query, its quality, and its impact on the plan, and the query runtime (including the overhead of NNs) as highlighted in Figure 2. We provide interesting IMDB queries and predefined NN models, so that visitors are able to pick a query over a context where there is a trained model. Visitors also have the opportunity to submit their own queries to the IMDB to explore with *PostCENN*. Then, our front-end reports the estimates, plans, and runtimes for each estimator. The user is also able to evaluate the overhead introduced by the NNs.

Lastly, there is the end-to-end comparison with the whole IMDB-benchmark testing the capability of *PostCENN*. There, a workload is executed under both estimators and the distribution of q-errors and runtimes of both approaches can be compared. This gives a large scale overview of the effectiveness of NNs as estimators by presenting statistical features, like mean or median, for a collection of q-errors and runtimes. This part is rounded off by a distribution chart for all q-errors in the benchmark.

Step 4 - Open Issues. Last but not least, we would like to discuss open issues with our demo visitors. For example, current research suggests that NNs are not always the superior estimator, but that histograms can sometimes outperform ML models [6]. This is due to the fact that NNs can introduce an overhead which cannot be compensated. With *PostCENN*, we can examine in depth which query and data properties lead to such cases. Furthermore, an adviser for when to use which estimator or the retraining of NNs if data changes are interesting open issues for further research.

4 SUMMARY

In this demo, we present *PostCENN*, an enhanced PostgreSQL database system with an end-to-end integration of machine learning (ML) models for cardinality estimation. Just like indexes to optimize the data access, we integrate ML models as a novel means to improve the accuracy quality of cardinality estimation for sub-parts of the database schema. *PostCENN* is a result of our research activities over the past few years. Moreover, *PostCENN* opens up future research topics like ML advisor or maintenance of ML models.

ACKNOWLEDGMENTS

This work was supported by the German Federal Ministry of Education and Research (BMBF, 01/S18026A-F) by funding the competence center for Big Data and AI “ScaDS.AI Dresden/Leipzig”.

REFERENCES

[1] Walter Cai, Magdalena Balazinska, and Dan Suciu. 2019. Pessimistic cardinality estimation: Tighter upper bounds for intermediate join cardinalities. In *SIGMOD*. 18–35.
[2] Andreas Kipf, Dimitri Vorona, Jonas Müller, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, Thomas Neumann, and Alfons Kemper. 2019. Estimating cardinalities with deep sketches. In *SIGMOD*. 1937–1940.
[3] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *PVLDB* 9, 3, 204–215.
[4] Henry Liu, Mingbin Xu, Ziting Yu, Vincent Corvinelli, and Calisto Zuzarte. 2015. Cardinality estimation using neural networks. In *ICSE*. 53–59.
[5] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S Sathiya Keerthi. 2019. An empirical analysis of deep learning for cardinality estimation. *arXiv preprint arXiv:1905.06425* (2019).
[6] Lucas Woltmann, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. 2020. Best of both worlds: combining traditional and machine learning models for cardinality estimation. In *aiDM@SIGMOD*. 1–8.
[7] Lucas Woltmann, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. 2021. Aggregate-based Training Phase for ML-based Cardinality Estimation. *BTW 2021* (2021).
[8] Lucas Woltmann, Claudio Hartmann, Maik Thiele, Dirk Habich, and Wolfgang Lehner. 2019. Cardinality estimation with local deep learning models. In *aiDM@SIGMOD*. 1–8.