Speeding Up End-to-end Query Execution via Learning-based Progressive Cardinality Estimation

Paper ID: 119

ABSTRACT

Fast query execution requires learning-based cardinality estimators to have *short inference time* (as model inference time adds to end-to-end query execution time) and *high estimation accuracy* (which is crucial for finding good execution plan). However, existing estimators cannot meet both requirements due to the inherent tension between model complexity and estimation accuracy. We propose a novel Learning-based Progressive Cardinality Estimator (LPCE), which adopts a *query re-optimization* methodology. In particular, LPCE consists of an initial model (LPCE-I), which estimates cardinality before query execution, and a refinement model (LPCE-R), which *progressively* refines the cardinality estimations using the actual cardinalities of the executed operators. During query execution, re-optimization is triggered if the estimations of LPCE-I are found to have large errors, and more efficient execution plans are selected for the remaining operators using the refined estimations provided by LPCE-R. Both LPCE-I and LPCE-R are light-weight query-driven estimators but they achieve both good efficiency and high accuracy when used jointly. Beside designing the models for LPCE-I and LPCE-R, we also integrate re-optimization and LPCE into PostgreSQL, a popular database engine. Extensive experiments show that LPCE yields shorter end-to-end query execution time than state-of-the-art learning-based estimators.

1 INTRODUCTION

A cardinality estimator estimates the size of the intermediate result tables in a query, and query optimizers rely on the estimated cardinalities to evaluate the costs of the execution plans [18]. Thus, cardinality estimators play a key role in finding a good execution plan and achieving short query execution time. Traditional approaches exploit statistics of the relation tables (e.g., via histogram or sampling) for cardinality estimation [2, 5, 21, 36]. Their estimation errors are large (e.g., many orders of magnitude) as they fail to account for the correlations among the tables [18, 21]. Recently, many learning-based cardinality estimators have been proposed [15, 30, 33, 40], and they significantly improve the accuracy of traditional methods. These methods can be classified into three categories [33, 34]: *data-driven*, *query-driven*, and *hybrid*. Data-driven estimators [11, 20, 31, 32, 35, 37, 39, 42] regard cardinality estimation as an unsupervised learning problem and model the joint distribution of the relation tables. Query-driven estimators [8, 10, 15, 22, 29, 30] treat cardinality estimation as a regression problem and map query content to cardinality values. Hybrid estimators [34] learn from both data and query for better accuracy. We will discuss more about these estimators in Section 2.

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

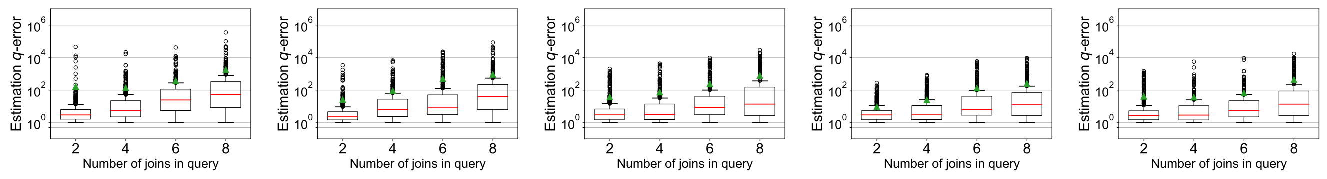
Table 1: Comparing some learning-based estimators.

	Name	Data access	q-error for estimation	Inference time (ms)
Hybrid	UAE [34]	Yes	5.05	23.4
Data-driven	DeepDB [11]	Yes	8.62	29.5
	NeuroCard [39]	Yes	6.39	23.2
	FLAT [42]	Yes	5.32	5.84
Query-driven	MSCN [15]	No	37.2	0.13
	TLSTM [30]	No	22.8	1.16
	Flow-Loss [22]	No	28.5	0.14
	LPCE	No	11.6	0.23

In this paper, we focus on query-driven estimators for three reasons. First, they are easy to deploy because they only require query samples and result cardinalities. Model training and inference can be out-sourced (e.g., to the could) as data are not involved. The training and inference costs do not have to scale with data size, and hence are more friendly to large-scale data. Model upgrade is also transparent to other functionalities of database systems (e.g., data access). Second, query-driven estimators do not access the relation tables, and thus are free from data security problems, which are crucial for domains such as finance and medicine [3]. Third, query-driven estimators typically have shorter inference time than data-driven and hybrid estimators (as shown in Table 1), and thus add less overhead to end-to-end query execution time.

To understand the performance of query-driven estimators, we conduct a preliminary test on the IMDB dataset [18]. We generate a set of queries with 8 joins following [15], and Table 1 reports the estimation error and inference time (for one single cardinality estimation and note that execution plan search requires many estimations). The results show that query-driven estimators have larger errors but shorter inference time than data-driven and hybrid estimators. This can be partially explained by the fact that query-driven estimators do not access the relation tables. To investigate the effect of query complexity (i.e., the number of joins), we generate a set of queries whose number of joins ranges from 2 to 8 and report the estimation error in Figure 1. The results show that the errors of query-driven estimators (as well as other estimators) are small for simple queries (e.g., with 2 or 4 joins) but become large for complex queries with more joins (e.g., >100x for 8 joins). The reason may be that the estimation errors propagate and amplify when more operators are involved. Guided by inaccurate estimations, the execution plan found by the query optimizer can be far from optimal, especially for complex queries.

The observations above inspire us to adapt query-driven estimators to the *query re-optimization framework* [4, 14, 24] with *progressive estimation*. In particular, the estimator should be able to



(a) MSCN [15] (b) TLSTM [30] (c) NeuroCard [39] (d) FLAT [42] (e) UAE [34]

Figure 1: Estimation error for queries with different number of joins. Red line and green triangle indicate median and mean, black rectangle shows 25th and 75th percentile, the black lines indicate 5th and 95th percentile, and circles are extreme errors.

progressively refine the cardinality estimations in the query execution process. This is possible because the exact cardinalities of the executed operators can be accessed, which directly influence intermediate result cardinality for the remaining operators. If the estimations made before query execution are found to have large errors, the initial execution plan may be bad, and thus a better execution plan can be selected using the refined estimations. In this way, we do not have to use complex models for accurate initial estimation (which leads to long inference time) but still obtain good execution plan. In fact, progressive estimation may be a general strategy to handle complex queries. Figure 1 shows that data-driven and hybrid estimators also have large errors for queries with many joins, and thus they can also lead to bad initial execution plan.

We provide one schematic illustration of progressive estimation in Figure 2, where the left panel shows the initial execution plan. After executing the operator at bottom of the join tree (i.e., $R_\sigma \bowtie T_\sigma$), we obtain its exact cardinality (i.e., 1,128), which enables more accurate cardinality estimations for sub-plans that involve the result of $R_\sigma \bowtie T_\sigma$ (e.g., $(R_\sigma \bowtie T_\sigma) \bowtie U_\sigma$ and $(R_\sigma \bowtie T_\sigma) \bowtie S_\sigma$). With the refined estimations, the optimizer finds that the plan in Figure 2(b) is better, whose *join order* (for table U_σ and S_σ) and *operator execution method* (for the join on the top) are different from the initial plan. However, enabling progressive estimation poses challenges on both effectiveness and efficiency. Specifically, effectiveness means that the estimator should fully utilize the information of the executed operators to provide more accurate estimations for the remaining operators. Efficiency means that the estimation model should have a short inference time as both initial estimation and progressive refinement adds to end-to-end query execution time.

To address the challenges above, we propose a framework named Learning-based Progressive Cardinality Estimation (LPCE). It consists of an initial cardinality estimation model (LPCE-I) and a progressive cardinality refinement model (LPCE-R). LPCE-I is designed to provide accurate initial estimation with low cost. In particular, LPCE-I utilizes a *node-wise loss function* to learn from the internal operators in an execution plan (instead of only the final query result as in existing estimators), which leads to high estimation accuracy. In addition, LPCE-I employs a *SRU-based model backbone*, and thus enables fast inference as SRU has fewer parameters than LSTM used in existing query-driven estimators and enjoys parallel matrix computation. Last but not least, we use a *knowledge distillation-based model compression* technique to further improve the inference efficiency of LPCE-I. LPCE-R is designed to progressively refine cardinality estimations for the remaining operators in the execution process. LPCE-R consists of three modules: one module is trained to extract the query contents of the executed operators, the second

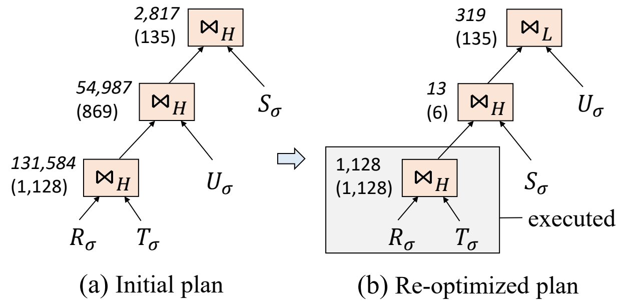


Figure 2: Query re-optimization example, upright number for real cardinality and italic number for estimated value. $\bowtie_L$ indicates nested loop join and $\bowtie_H$ is hash join.

module focuses on the exact cardinalities of the executed operators, while the third module fuses the information provided by the former two modules to conduct estimation for the remaining operators. All three modules adopt the lightweight structure of LPCE-I, and thus LPCE-R also enjoys efficient inference.

We also integrate query re-optimization and LPCE into PostgreSQL, a popular database engine. As PostgreSQL employs pipelined query processing for efficiency, and checkpoints are carefully placed at appropriate operators to detect large cardinality estimation errors during execution and trigger re-optimization with low overhead.

To summarize, we made the following contributions in this work:

- We observe a tension between inference time and estimation accuracy in existing learning-based cardinality estimators. To resolve this tension, we propose progressive estimation, which is the first to adapt learning-based estimators for query re-optimization and allows both fast inference and accurate estimation. Progressive estimation may also be a general strategy to handle the large estimation errors for complex queries (Section 1).
- We design LPCE to instantiate the progressive estimation idea with query-driven cardinality estimators. LPCE consists of an initial estimation model (LPCE-I) and a progressive refinement model (LPCE-R). Tailored designs are introduced to ensure that LPCE-I achieves accuracy and efficiency (Section 4), and LPCE-R effectively extracts information from the executed sub-plans to refine cardinality estimations (Section 5).
- We integrate LPCE into PostgreSQL (Section 6) and conduct extensive experiments to compare LPCE with state-of-the-art learning-based estimators (Section 7). The results show that LPCE outperforms recent query-driven, data-driven and hybrid estimators in end-to-end query execution time because of its fast inference and good accuracy enabled by progressive estimation.

2 RELATED WORK

Learning-based cardinality estimators: Recently, the database community recognized the potential of replacing traditional cardinality estimators with learning-based models (e.g., neural network and autoregressive model) [16, 33]. We classify existing learning-based estimators into three categories, i.e., *query-driven*, *data-driven* and *hybrid*, and discuss some representatives in each category. More comprehensive discussions can be found in the surveys [9, 31, 33].

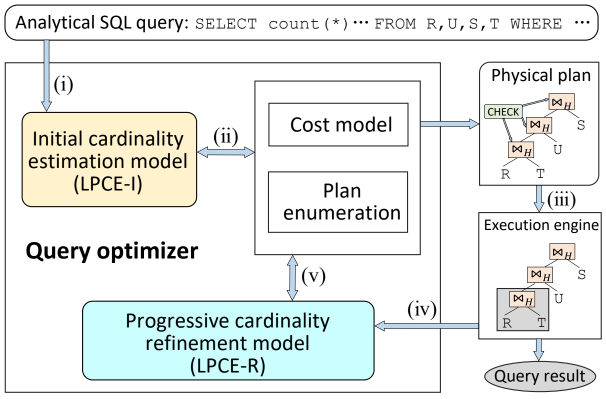
Data-driven estimators [11, 20, 31, 32, 35, 37, 39, 40, 42] try to capture the correlations and distributions of data across the relation tables. In particular, DeepDB [11] adopts the relational sum product network (RSPN) to model the joint probability distribution of the relations, and estimates cardinality by computing probabilities and expectations based on RSPN. NeuroCard [39] trains a single deep autoregressive model using samples collected from the full outer join results for all relations, and can estimate cardinality for queries that join any subset of the relations. FLAT [42] utilizes the factorize-split-sum-product network (FSPN) [38], which is based on the graphical model, and models the joint probability density function (PDF) of the attributes. Glue [43] learns to decouple the correlations among the tables and merge cardinality results of individual tables to predict the cardinality of join results. FACE [32] adopts a model based on normalizing flow [27] and learns a continuous joint distribution for the tables.

Query-driven estimators [8, 10, 15, 22, 29, 30] formulate cardinality estimation as a regression problem. The contents of queries (e.g., filter conditions and joined tables) are embedded as tensor inputs and the true cardinalities of the queries are used as target outputs to learn a mapping from queries to cardinalities. Ref. [7] uses regression techniques such as XGBoost to produce approximate cardinality labels. MSCN [15] transforms a query into a feature vector and uses a multi-set convolution network [41] to map it to cardinality estimation. TLSTM [30] processes a query execution plan recursively following its tree structure using LSTM (a kind of RNN) [13]. Instead of matching the true cardinalities, Flow-loss [22] trains the cardinality estimation model such that it leads to accurate estimates for the costs of execution plans.

Hybrid estimator learns from both relation tables and query samples. UAE [34] adopts a unified deep autoregressive model, which learns the joint distribution of the tables in an unsupervised manner and is trained to use query contents as auxiliary information in a supervised manner.

As discussed in Section 1, existing learning-based estimators cannot achieve *fast inference* and *high accuracy* at the same time, which are crucial for short end-to-end query execution time. With progressive estimation, our LPCE enjoys the fast inference of query-driven estimators and can progressively refine cardinality estimations to remedy large errors. As data-driven and hybrid estimators also have large estimation errors for complex queries, they may also benefit from the progressive estimation strategy.

Query re-optimization: Accurate cardinality estimation is challenging, and thus re-optimization is proposed to combat the influence of large estimation errors on query optimization [1, 4, 6, 14, 25, 36]. Existing re-optimization techniques can be classified into two categories: (1) *Re-optimization before query execution.* Ref. [36] uses a sampling-based method to evaluate cardinality estimation errors before running the execution plan and adjusts the execution plan if the errors are found to be large. (2) *Re-optimization during query execution.* Ref. [1, 14, 24] sample tuples from intermediate execution results, and tries different operator implementations and join orders on the samples to adjust the remaining execution plan. Our LPCE differs from existing query re-optimization techniques in that it uses machine learning models to extract information from the executed sub-queries to refine cardinality estimations. To our knowledge, we are the first to utilize learning-based estimators for query re-optimization.



Analytical SQL query: SELECT count(*) ··· FROM R,U,S,T WHERE ···
(i)
Initial cardinality estimation model (LPCE-I)
(ii)
Cost model
Plan enumeration
Query optimizer
Physical plan
CHECK
R
T
U
S
(iii)
Execution engine
(v)
(iv)
Progressive cardinality refinement model (LPCE-R)
Query result

Figure 3: End-to-end query execution with LPCE.

3 OVERVIEW OF THE LPCE FRAMEWORK

We consider *select-project-equijoin-aggregate* [15, 23] queries in the following form:

$$\begin{aligned}&\text{SELECT COUNT}(*)\\&\text{FROM R, U, S, T}\\&\text{WHERE } \mathsf{R}_\sigma = \mathsf{U}_\sigma \text{ AND } \mathsf{U}_\sigma = \mathsf{S}_\sigma \text{ AND } \mathsf{S}_\sigma = \mathsf{T}_\sigma,\end{aligned}$$

where R, U, S and T are relation tables, σ represents the filtering predicates (e.g., R.$a < 10$, which returns tuples in table R whose attribute a is smaller than 10) and the filtering operator can be mathematical (e.g., < and =) or complex predicate (e.g., IN and LIKE) [30]. For a query, the query optimizer finds an efficient execution plan (usually expressed as a join tree) by enumerating possible execution plans using algorithms such as dynamic programming [18]. The cost of a plan is calculated based on cardinality estimations of its sub-plans (e.g., $\mathsf{R}_\sigma \bowtie \mathsf{T}_\sigma$ and $(\mathsf{R}_\sigma \bowtie \mathsf{T}_\sigma) \bowtie \mathsf{U}_\sigma$ for the plan in Figure 2(a)).

3.1 Query Execution with LPCE

LPCE aims to reduce the end-to-end execution time of queries and consists of an initial estimation model LPCE-I and an estimation refinement model LPCE-R. Figure 3 shows how LPCE works in the query re-optimization framework.

i. When a query is submitted, it is sent to LPCE-I for initial cardinality estimations of all its possible sub-plans;
ii. Using the initial estimations, the query optimizer chooses a good execution plan based on its plan search algorithm;
iii. The chosen plan is executed by the database, and checkpoints are placed at some operators to monitor if the initial estimations are significantly different from the actual cardinalities;

iv. Once large estimation error is detected, query execution is paused and LPCE-R is invoked to refine the cardinality estimations for the feasible sub-plans of the remaining operators;
v. Based on the refined estimations, the query optimizer searches a better execution plan for the remaining operators.

The idea behind LPCE is simple: during the query execution process, the result tables of the executed sub-plans are available, which serve as inputs to the operators that remain to be executed and thus provide crucial information to refine their cardinality estimations. LPCE is the most beneficial to complex and long running queries, for which query-driven estimators can have large estimation errors and thus the initial execution plan may be far from optimal.

3.2 Design Goals of LPCE

To achieve short end-to-end query execution time, LPCE needs to meet three key goals, i.e., *high accuracy, progressive refinement and fast inference*, which we elaborate as follows.

High Accuracy: Query re-optimization with LPCE-R incurs overheads that add to end-to-end query execution time. To reduce the frequency of re-optimization, the initial estimation model LPCE-I should provide accurate cardinality estimations such that the query optimizer can find a good initial plan. For this goal, we design the node-wise loss function in Section 4.

Fast inference: The inference time of both LPCE-I and LPCE-R (if re-optimization is triggered) counts as part of the end-to-end query execution time. However, query optimization needs to estimate the cardinality of many sub-plans and inference for machine learning models can be expensive. Thus, LPCE models should have a short inference time in order to achieve short end-to-end query execution time. To this end, we use a light-weight model backbone for LPCE and further compress it via knowledge distillation in Section 4.

Progressive refinement: To ensure that query re-optimization can find good execution plans, LPCE-R should effectively reduce estimation error when more operators are executed. For this purpose, LPCE-R should fully utilize information of the executed sub-plans, especially their real cardinalities. To achieve this goal, we design a model structure with three modules for estimation refinement in Section 5, which explicitly considers the executed sub-plans.

There are many other important problems for learning-based cardinality estimators, e.g., handling data updates, reducing model training time and coping with out-of-distribution queries. We leave them for future work and do not consider them in our design goals.

4 CARDINALITY ESTIMATION MODEL

In this part, we first introduce the generic pipeline for learning-based query-driven cardinality estimation models as background (Section 4.1), which we also follow in LPCE. Then we present the key designs of our initial cardinality estimation model (i.e., LPCE-I), including node-wise loss function, SRU-based light-weight model and knowledge distillation assisted model compression (Section 4.2~4.4), which lead to high accuracy and fast inference.

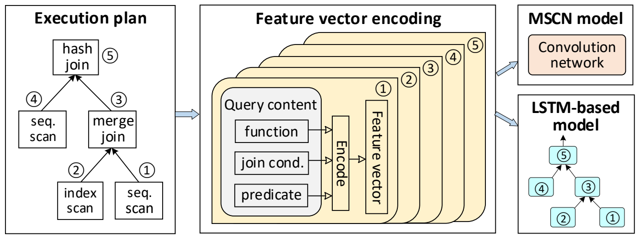


Figure 4: Workflow of learning-based estimation model.

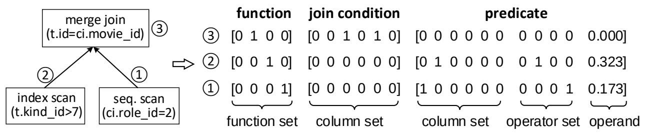


Figure 5: Example for feature encoding.

4.1 Model Learning Pipeline

Given an execution plan, learning-based cardinality estimation models predict the cardinalities of the result tables (both indeterminate and final) [8, 10, 15, 30, 39]. As illustrated in Figure 4, training learning-based cardinality estimation models takes three steps, i.e., *sample collection, feature encoding and model learning*, which we elaborate as follows.

Sample collection: Cardinality estimation is usually formulated as a regression problem, in which the execution plan is the input and the cardinalities are the output. Training samples can be collected from historical execution log. In the case of new database at "cold start", sample queries can be randomly generated according to the relational graph of the underlying dataset [15] and the execution plans can be obtained from database engines (e.g., via the 'EXPLAIN QUERY' command in PostgreSQL). As shown in Figure 4, an execution plan is usually a tree, in which each node represents an operator (e.g., hash join, index scan and filter). Each node also contains detailed information about the query, such as filtering predicate (e.g., $t.kind\_id > 7$) and join condition (e.g., R.a = U.a). We can execute the plans and obtain the cardinality of each node using the 'EXPLAIN ANALYZE QUERY' command or by adding counters.

Feature encoding: As machine learning models usually take vector input, the nodes in an execution plan are encoded into a feature vector during prepossessing. Following existing works [15, 30], we encode the *function, join condition, predicate* of each node as illustrated in Figure 5. Function is the logical operator at a node (e.g., join, scan) and we encode it as a one-hot vector with length $|P|$, where P is the set of all possible operators. Note that we consider function as logical operator rather than physical operator such as hash join and index scan. This is because cardinality estimation is done before physical operator chosen during plan generation. Join condition are the columns that are joined and we encode it as a two-hot vector with length $|C|$, where C is the set of all table columns for the dataset. For example, the join condition of node 3 is $[0, 0, 1, 0, 1, 0]$ in Figure 5, which indicates that the attribute value of column 3 needs to be equal to column 5 in the join. For nodes that do not conduct join, the join condition is a zero vector. *Predicate* is in the form of $[column_{id}, operator_{id}, operand]$ (e.g., $t.kind\_id > 7$), where $column_{id}$ and $operator_{id}$ use one-hot encoding while the

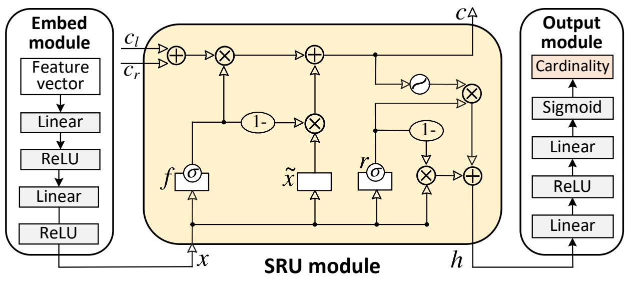


Figure 6: The structure of the SRU-based model in LPCE-I.

value of *operand* is recorded as float after normalization. We concatenate the function, join condition, predicate vectors of a node to obtain a single feature vector and readers can refer to [15, 30] for more details about feature encoding.

Model learning: MSCN [15] and TLSTM [30] are two state-of-the-art models for query-driven cardinality estimation. MSCN stacks the feature vectors of all nodes in an execution plan and uses a multi-set convolution network [41] to map them to cardinality estimation. TLSTM processes an execution plan recursively following its tree structure using LSTM [13]. The embedding of the root node is treated as the query representation and used to predict cardinality. MSCN has large estimation errors as it does not utilize the structure of the execution plan while TLSTM suffers from the high computation complexity of LSTM. Our LPCE-I also processes an execution plan recursively using RNN as in TLSTM but designs are introduced to boost both accuracy and efficiency.

4.2 SRU-based Light Weight Model

Our SRU-based model is illustrated in Figure 6, which is used to process each node in an execution plan. The model consists of three modules, i.e., *embed module, SRU module and output module*. The embed module is used to map the sparse feature embedding of a node (introduced in Section 4.1) to dense embedding x. We use a two-layer fully-connected neural network with ReLU activation function as the embed module. The output module is used to map the node representation h (which encodes the sub-plan rooted at the node) to cardinality estimation, which is also two-layer fully-connected neural network. The final layer of the output module uses the sigmoid activation function to generate a float in [0,1], which is interpreted as the ratio of the estimated cardinality over the maximum cardinality observed in the train set.

For a node in an execution plan, the SRU module takes its query content embedding x, the encodings of its left and right child in the execution plan tree (i.e., c_l and c_r), as input to generate the node encoding c and node presentation h. The SRU module is an RNN as node encoding c is passed to its parent node, which reuses the SRU module with the same parameter. The SRU module uses the simple recurrent unit (SRU) [17] as the model, which conducts computation following

$$\begin{aligned}\tilde{x} &= W_x x\\ f &= \rho(W_f x + b_f)\\ r &= \rho(W_r x + b_r)\\ c &= f \odot (c_l + c_r) + (1-f) \odot \tilde{x}\\ h &= r \odot tanh(c) + (1-r) \odot x\end{aligned} \tag{1}$$

where W_x, W_f and W_r are parameter matrices, b_f and b_r are bias vectors, $\odot$ denotes element-wise multiplication and ρ is the activation function sigmoid. f is the forget gate, which controls the weight of the children encodings (i.e., c_l and c_r) and projected node embedding (i.e., $\tilde{x}$) when generating encoding c for the current node. r is the reset gate, which uses the node embedding x and node encoding c to generate node representation for cardinality estimation.

We choose SRU as the RNN model in LPCE-I as it has higher computation efficiency than the LSTM in TLSTM. LSTM needs 8 matrix multiplications while SRU needs only 3. In addition, the 3 matrix multiplications (to compute $\tilde{x}$, f and r, respectively) in SRU can be parallelized while the matrix multiplications in LSTM have data dependencies. In the experiments, we show that our SRU-based model is smaller in memory consumption and 1.7x faster in inference speed compared with TLSTM. We also show that changing from LSTM to SRU has negligible loss in estimation accuracy.

4.3 Node-wise Loss Function

Both MSCN and TLSTM use the mean q-error as the loss function, which is defined as

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n} q_i \quad \text{with } q_i = \frac{\max(c_i, \tilde{c}_i)}{\min(c_i, \tilde{c}_i)}, \tag{2}$$

where c_i and $\tilde{c}_i$ are the real and model estimated cardinalities for the final result table of the i^{th} execution plan, and the train set contains n plans. q_i is the q-error of the i^{th} plan, which measures the estimation accuracy and lower value indicates better estimate (note that $q_i \geq 1$). As Equation (2) only considers estimation errors for the final result of each query, we call it the *query-wise loss function*. In LPCE-I, we use the *node-wise loss function* defined as

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{m_i} q_{ij} \quad \text{with } q_{ij} = \frac{\max(c_{ij}, \tilde{c}_{ij})}{\min(c_{ij}, \tilde{c}_{ij})}, \tag{3}$$

where c_{ij} and $\tilde{c}_{ij}$ are the real and model estimated cardinality for the j^{th} node in the i^{th} execution plan, and m_i is the number of nodes in the i^{th} execution plan. Different from the query-wise loss function, the node-wise loss function considers the q-error of all nodes in each execution plan.

We observe that the node-wise loss function significantly improves estimation accuracy and the reasons are two-fold. First, it is a kind of data argumentation that enlarges the train set. For example, for a single execution plan $(A \bowtie B) \bowtie (C \bowtie D)$ in the train set, the node-wise loss function actually uses the estimation error of three execution plans, i.e., $(A \bowtie B)$, $(C \bowtie D)$ and $(A \bowtie B) \bowtie (C \bowtie D)$. As complex queries contain many internal nodes, the data argumentation effect is significant. Second, it allows supervision for every node in the execution plan. TLSTM (and our LPCE-I) uses RNN to embed an execution plan from leaf to root along its tree structure and the query-wise loss function only provides supervision for the root node, which means that the gradients need to back-propagate in time to reach the internal nodes. By providing direct supervision signal for the internal nodes, the node-wise loss function produces more informative embedding for the internal nodes, which yields more accurate representation and thus cardinality estimation for the entire query. We observe that the gain of the node-wise loss

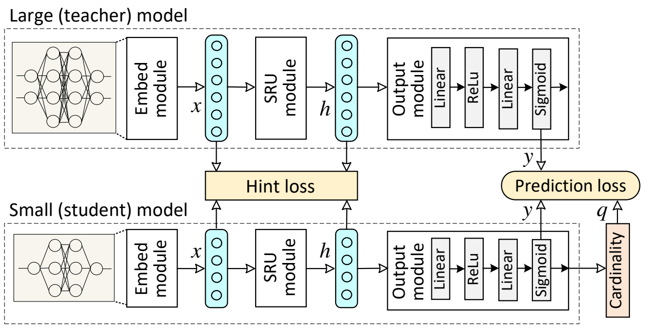


Figure 7: Model compression via knowledge distillation.

function is significant for complex execution plans with a deep tree structure (Section 7).

4.4 Knowledge Distillation for Compression

The model structure parameters of LPCE-I (e.g., the size of the embedding vectors and number of hidden units in the neural networks) control the complexity and accuracy of the model. Although using a small model provides fast inference, we found directly training the small model yields poor accuracy. This is because smaller models have weaker ability to learn and generalize. To train small model with high accuracy, we use *knowledge distillation* [12, 28], which uses a large (thus accurate) *teacher model* to guide the learning of the small *student model*. The idea is that the student model can learn useful knowledge by matching the output of the teacher model as shown in Figure 7, which is easier to fit than training data as it is produced by the teacher model whose structure is the same as the student model.

To conduct knowledge distillation, we first train a teacher model with high complexity and accuracy. Then, we train the student model using the following hint loss

$$\mathcal{L}_{\text{hint}} = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{m_i} \|x_{ij}^t - p_e(x_{ij}^s)\|_1 + \|h_{ij}^t - p_s(h_{ij}^s)\|_1. \tag{4}$$

For the jth operator in the ith execution plan, x_{ij}^t and x_{ij}^s are the output of the embed module of the teacher model and student model, respectively. Similarly, h_{ij}^t and h_{ij}^s are the node representation produced by SRU module of the teacher model and student model. $p_e(.)$ and $p_s(.)$ are two single-layer neural networks that are used to adjust the outputs of the student model to the same size as the teacher model. After that, we further calibrate the student model using the following prediction loss

$$\mathcal{L}_{\text{predict}} = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{m_i} \alpha q_{ij}^s + (1-\alpha)\left|y_{ij}^t - y_{ij}^s\right|, \tag{5}$$

where q_{ij}^s is the q-error for the cardinality estimation of the student model, y_{ij}^t and y_{ij}^s are the logit before the `sigmoid` activation function in the output module of the teacher and student model. We train the student model to fit the logit of the teacher model as it has a direct impact on cardinality estimation. α is a weight used to balance the two loss terms and usually set as 0.5. In the experiments, we show that knowledge distillation can compress the LPCE-I model by more than 10x and speed up inference by 1.8x without degrading accuracy.

5 CARDINALITY REFINEMENT MODEL

Query-driven estimator produces cardinality prediction at large errors for queries that model has not learned. We propose to query re-optimization with cardinality refinement to combat the large errors and adjust the execution plan to be better one. In this part, we present the design of LPCE-R, our model for progressive cardinality estimation refinement.

5.1 The Model Structure of LPCE-R

As illustrated in Figure 8, query execution in database systems is conducted from leaf to root along the execution plan tree. At some points in time, some nodes in the query plan (shadowed in Figure 8) are finished while some remain to be executed. The idea of LPCE-R is to use the information of the finished nodes to refine cardinality estimations for the remaining nodes as the cardinalities of the remaining nodes depend on the intermediate result tables of the finished nodes. Thus, LPCE-R needs to solve three problems: (i) what information should we extract from the executed sub-plan, (ii) how to effectively exploit the information for cardinality refinement and (iii) how to efficiently refine the cardinality estimations in the query execution process.

To address the problems above, LPCE-R adopts a hybrid structure with three modules as illustrated in Figure 8. All three modules adopt the same structure as LPCE-I in Section 4.2 but are trained differently (will be discussed in Section 5.2). Cardinality module and content module are used to extract different information from the executed sub-plan. A connect layer merges the information from cardinality module and content module as input for refine module, which refines the cardinality estimation for the remaining nodes. In the following, we introduce the design of each of these components.

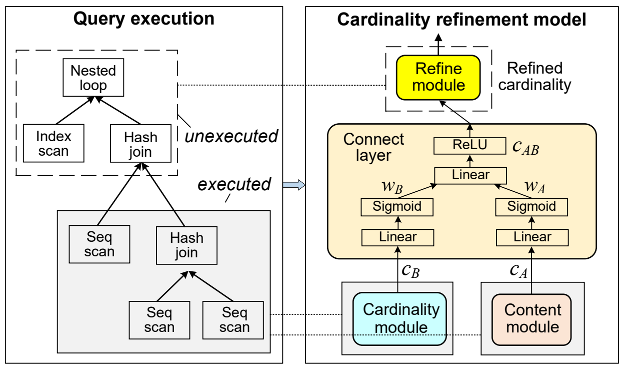


Figure 8: An overview of LPCE-R for estimation refinement.

Information extraction with two modules: Two kinds of information can and should be extracted from the executed sub-plan, i.e., *real cardinality* and *query content*. For example, if the cardinality of a node is estimated as 10 originally but the real cardinality is found to be 1000 after finishing it. Noticing this 100x underestimation is crucial for refining the estimations for the remaining nodes as the results of these nodes depend on the finished nodes. In addition, the semantics of the executed sub-plan (i.e., query content), such as the type of the operators, joined columns and filtering predicates, are also important for estimating the cardinalities of the remaining nodes. For example, a filtering predicate in the executed sub-plan (e.g., $R.a < 10$) could have a direct impact on the remaining join operators (e.g., $R \bowtie U$ with $R.a = U.a$). Therefore, cardinality module is used to extract the real cardinality of the executed sub-plan and adopts the structure of LPCE-I. Content module is used to extract the query content and its structure is also exactly the same as LPCE-I. One can append the real cardinalities of its two children to the query content feature encoding of an executed node, which is used as input for the single module. However, in the experiments, we observed using both cardinality and content module yields significantly higher accuracy than using only one module, which suggests that both real cardinality and query content are important for estimation refinement.

Learned information merge: Both cardinality and content module produce an embedding (i.e., c_A and c_B), which encodes the executed sub-plan. As shown in Figure 8, we train a connect layer to merge the two embedding as input for refine module. The connect layer uses a single-layer neural network with `sigmoid` activation function to learn the merge weights for c_A and c_B, respectively. The weighted combination of c_A and c_B is processed by a single-layer neural network with ReLU activation function. Specifically, the connect layer conducts the following computation

$$\begin{aligned} w_A &= \rho(W_A c_A + b_A) \\ w_B &= \rho(W_B c_B + b_B) \\ c_{AB} &= \text{ReLU}\big(W_{AB}(w_A \odot c_A + W_B \odot c_B) + b_{AB}\big), \end{aligned} \tag{6}$$

where $\rho(\cdot)$ is the `sigmoid` function and $\odot$ denotes element-wise multiplication. W_A, W_B and W_{AB} are parameter matrices. We use a learned connect layer because we observed empirically that the contributions of real cardinality (i.e., c_B) and query content (i.e., c_A) to accuracy vary in different scenarios. For example, when the number of remaining operators is small, using only real cardinality already provides good estimation accuracy. However, the query content is important when there are many remaining operators due to more dependencies on the semantics of the finished sub-plan. The connect layer can learn to adjust the weights of real cardinality and query content according to the scenario.

Efficient progressive refinement: Refine module is used to refine the cardinality estimations of the remaining operators and its structure is exactly the same as LPCE-I. The embedding of the executed sub-plan (i.e., c_{AB}) is fed as input to the SRU of refine module, which replaces both or either of c_l and c_r (see Figure 6) according to the execution situation. When an operator finishes, both cardinality and content module update their embedding according to the previously executed sub-plan (which has already been processed by the modules) and the just finished operator. Using the updated embedding, refine module refines the estimations for all remaining operators. Progressive refinement is efficient as cardinality and content module do not need to process the entire finished sub-plan from scratch. Instead, the embedding is updated incrementally with each finished operator.

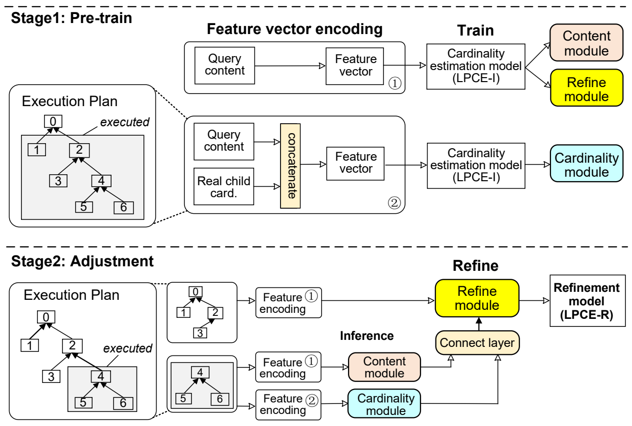


Figure 9: The training workflow of LPCE-R.

5.2 Training Procedures

As illustrated in Figure 9, the training of LPCE-R consists of two stages, i.e., *pre-train* and *adjustment*. In the pre-train stage, content module is trained in the same way as LPCE-I and refine module shares the same parameters as content module. For cardinality module, we concatenate the feature encoding (for the query content) of each operator with the real cardinalities of its two children as input, and train the module to minimize the node-wise loss function in Equation (3). Note that for the leaf nodes in an execution plan, their real cardinalities are the number of tuples in the considered attributes.

In the adjustment stage, we froze cardinality and content module, and fine-tune refine module. In this case, an execution plan with m operators provides m-1 training samples for refine module. Specifically, when each operator finishes, content and cardinality module are used to obtain the embedding of the executed sub-plan, and refine module is trained to predict the cardinalities of the remaining operators. The loss function is also the node-wise loss function in Equation (3). We provide such an example in the lower part of Figure 9, in which operators 4, 5, 6 are executed and the cardinalities of operators 0 and 2 need to be re-estimated.

6 QUERY OPTIMIZATION WITH LPCE

In this part, we show how to integrate LPCE into database engines. First, we discuss how a query optimizer can use LPCE-I to find good execution plan *before* runtime. Then, we explain how to conduct query re-optimization with LPCE-R *during* runtime. Our discussions are based on PostgreSQL but the techniques generalize to other database engines such as MySQL and SQLite.

6.1 Query Initial Optimization with LPCE-I

The query optimizer of PostgreSQL uses a dynamic programming procedure to enumerate possible execution plans in order to find the plan with the minimum cost. For a query that joins n relation tables, enumeration starts at level-1 for the base tables (e.g., R_σ, along with possible filtering predicates) and ends at level-n for the entire query (e.g., $\mathsf{R}_\sigma \bowtie \mathsf{T}_\sigma \bowtie \mathsf{U}_\sigma$). At level-$i$, a sub-query joins i relations and requires cardinality estimations of its children queries (which join some of the relations in the sub-query) at lower levels to find the best execution plan. For example, query $\mathsf{R}_\sigma \bowtie \mathsf{T}_\sigma \bowtie \mathsf{U}_\sigma$ at

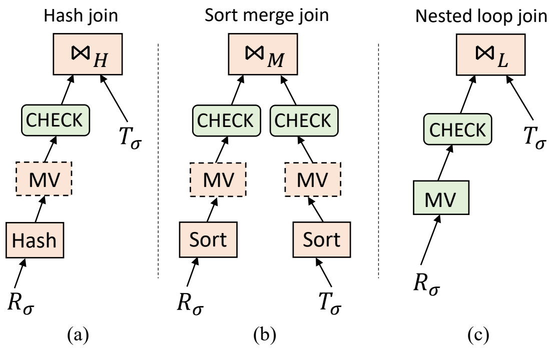


Figure 10: Modifications of the join operators to support query re-optimization, color brown indicates the original operator logic in PostgreSQL while color green indicate our modifications. 'MV' means tuple materialization.

level-3 requires cardinality estimations of R_σ, T_σ and U_σ at level-1, and $\mathsf{R}_\sigma \bowtie \mathsf{T}_\sigma$, $\mathsf{T}_\sigma \bowtie \mathsf{U}_\sigma$ and $\mathsf{R}_\sigma \bowtie \mathsf{U}_\sigma$ at level-2.

We replace the histogram-based cardinality estimator of PostgreSQL with LPCE-I for more accurate cardinality estimations. LPCE-I inferences for all sub-queries on the same level are conducted in a batch as they have the same number of input relations and the feature vectors of a sub-query are small. To avoid repetitive computation, cardinality estimation results of sub-queries are stored in a *memory pool* for lookup following [30].

For PostgreSQL, query execution time can be decomposed into planning time T_P and execution time T_E as its histogram-based cardinality estimator has negligible overhead. With learning-based estimators, end-to-end query execution time can be expressed as

$$T_{end} = T_P + T_I + T_E, \tag{7}$$

where T_I is the model inference time for cardinality estimation. For queries with a short execution time T_E, T_I can dominate T_{end} if the model is complex. In the experiments in Section 7, we show that LPCE-I yields shorter T_{end} than existing data-driven estimators in most cases due to its simple model and short inference time.

6.2 Query Re-optimization LPCE-R

Conceptually, query re-optimization works as follows. When each operator finishes, we use q-error to measure the difference between the actual cardinality and initially estimated cardinality of its output relation, and trigger query re-optimization if the error is larger than a threshold (empirically set as 50). In this case, LPCE-R is used to refine cardinality estimations, and the optimizer searches the optimal execution plan among those that continue from the current progress and those that restart query processing from scratch. To allow query execution to start from the executed sub-plans, the intermediate relations are materialized. However, PostgreSQL does not provide native support for cardinality estimation error check, and adopts pipelined query processing when possible. A pipeline involves several operators, and the tuples produced by one operator are passed to the downstream operators for processing without materialization.

To support query re-optimization, we use checkpoint to assert cardinality estimation errors following [24], which is a simple counting operator that collects the output cardinality for an operator. We place checkpoint after operators that materialize intermediate tuples (e.g., *sort* and *materialization* in PostgreSQL) as they block pipelined processing and buffer the tuples. We also place checkpoint at join operators, and PostgreSQL has three join implementations, i.e., *hash join*, *merge join*, and *nested loop join*. As shown in Figure 10(a) and (b), checkpoint (CHECK) is placed after building hash table on the inner side (the smaller table) for hash join, and after the sorting on both sides for merge join. As hash join and merge join both materialize tuples in PostgreSQL (the dotted 'MV' in Figure 10), no extra materialization overhead is inured. However, both sides are pipelined for nested loop join. As shown in Figure 10(c), we block the pipeline and materialize the tuples after scanning the table on the outer side (the smaller table). To test the overhead introduced to nested loop join, we ran 500 queries from the IMDB benchmark. The query execution time and the peak memory comsuption are increased by 1.2% and 5.8%, respectively. The overhead is small because nested loop join is used only when the cardinality of the outer side is small. More details about the implementation of re-optimization can be found in the appendix.

For a query that triggers re-optimization, the end-to-end execution time can be expressed as

$$T_{end} = T_P + T_I + T_R + T_E, \tag{8}$$

where T_R is the re-optimization time, which includes the plan search and model inference time during re-optimization. Thus, re-optimization should be triggered when its execution time reduction outweighs T_R. Currently, we use a simple q-error threshold, which may invoke unnecessary re-optimizations or miss re-optimization opportunities. The experiments in Section 7 show that this simple rule already achieves good performance and we leave designing more sophisticated policies to trigger re-optimization as future work. Another subtlety is that re-optimization may be triggered multiple times for a query, and we limit the maximum re-optimization to 3 times to control the overhead of re-optimization. We observed that are a small number of difficult queries, for which the estimation error of LPCE-R is large even with information from the executed sub-plans. This may be because LPCE-R does not learn these queries from the training samples, and conducting re-optimization many times for them is not beneficial.

7 EXPERIMENTAL EVALUATION

We present our empirical studies in this part. Section 7.1 introduces the experiment settings. Section 7.2 evaluates the end-to-end query execution time of LPCE and compares with state-of-the-art baselines. Section 7.3 validates the effectiveness of LPCE's key designs.

7.1 Experiment Settings

Workloads: We use the IMDB dataset [18] for the experiments, which contains 22 tables recording facts (e.g., actors, directors and companies) about over 2.1M movies. IMDB is widely utilized in related researches [15, 16, 19, 30, 39, 40] as a challenging benchmark for cardinality estimation due to its non-uniform data distributions and complex correlations among the tables. We generate the train

Table 2: Percentiles of end-to-end execution time reduction compared with PostgreSQL (large the better, best marked in bold).

Join-six	5th	25th	50th	75th	95th
DeepDB	-232%	7.85%	56.3%	88.5%	98.1%
NeuroCard	-115%	11.9%	45.9%	71.8%	97.4%
FLAT	-43.7%	54.9%	75.7%	92.9%	99.1%
UAE	-116%	44.7%	64.4%	89.7%	98.9%
MSCN	-240%	29.6%	67.5%	91.0%	99.4%
Flow-Loss	-136%	35.6%	70.6%	91.8%	99.4%
TLSTM	-118%	38.5%	72.2%	92.3%	98.0%
LPCE-I	-13.4%	52.3%	77.5%	94.1%	99.6%
LPCE-R	**-5.21%**	**68.2%**	**86.9%**	**96.4%**	**99.7%**

Join-eight	5th	25th	50th	75th	95th
DeepDB	-442%	-22.5%	22.8%	71.7%	95.5%
NeuroCard	-370%	-14.3%	29.6%	74.5%	95.7%
FLAT	-65.2%	62.4%	78.5%	92.3%	97.7%
UAE	-360%	-8.85%	35.2%	75.8%	95.8%
MSCN	-986%	19.5%	63.6%	81.2%	98.1%
Flow-Loss	-453%	28.2%	67.9%	84.6%	98.8%
TLSTM	-384%	32.5%	70.8%	90.5%	97.2%
LPCE-I	-28.6%	58.8%	73.9%	95.7%	99.3%
LPCE-R	**-9.87%**	**74.5%**	**86.6%**	**96.0%**	**99.3%**

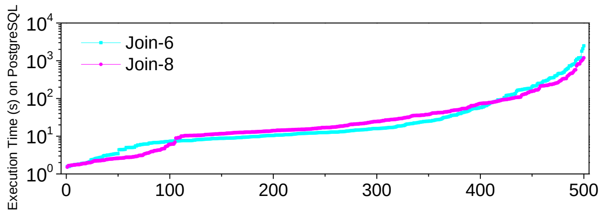
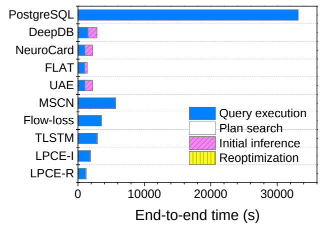
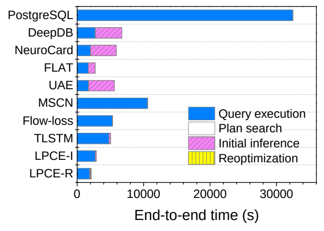


Figure 11: Execution time of the test queries on PostgreSQL.



(a) *Join-six*



(b) *Join-eight*

Figure 12: Decomposition of end-to-end query execution time, statistics are aggregated over the 500 test queries.

and test queries according to the relational graph of the tables following [15]. The train set contains 10,000 sample queries with 6-8 joins, and 10% of these queries are randomly selected as the validation set. We used two query sets for performance test: (1) *Join-six*, which contains 500 queries with 6 joins; and (2) *Join-eight*, which contains 500 queries with 8 joins. As shown in Figure 11, we select the test queries such that their end-to-end execution time on PostgreSQL spreads over a wide range (i.e., from 1s to 1,500s) to consider queries with different complexity. Before executing the test queries on PostgreSQL, we run ANALYZE for all tables and 100 sample queries for warming up.

Baselines: We compare our LPCE with state-of-the-art learning-based cardinality estimators that are discussed in Section 2. The baselines include: (1) three query-driven estimators, i.e., MSCN [15], TLSTM [30], and Flow-Loss [22]; (2) three data-driven estimators, i.e., DeepDB [11], NeuroCard [39], and FLAT [42]; and (3) one hybrid estimator, UAE [34]. Glue [43] and FACE [32] are not compared as they are not open-source. Our methods include LPCE-I, which only conducts cardinality estimation before query execution, and LPCE-R, which uses LPCE-I for initial estimation but may trigger query re-optimization and progressive estimation refinement. We use end-to-end query execution time as main performance metric.

Implementation details: For MSCN, Flow-Loss, DeepDB, FLAT, and NeuroCard, we use their open-source implementations and adopted the hyper-parameters recommended by their authors. For UAE, we use the implementation kindly provided by its authors. Since TLSTM is not open-source, we implemented it on our own and tuned the hyper-parameters to reproduce the results reported in its paper. As MSCN and DeepDB do not support range queries on categorical string columns, we encode these columns into integers using dictionary encoding.

For our LPCE-I, the number of hidden units for the embedding module, SRU module and output module are 64, 196 and 1024, respectively. LPCE-I is compressed from a large model via knowledge distillation, for which the number of hidden units for the embedding module, SRU module and output module are 256, 1024 and 1024, respectively. For LPCE-R, query reoptimization is triggered when the q-error between the actual cardinality of an intermediate result table and the initial estimation is above 50. We limit the maximum number of re-optimization to 3 times. The *tuplestore* used to buffer the outer tuples of nested loop join for query re-optimization is set as 20 MB, which is the default size of *tuplestore* for PostgreSQL. The models are trained with a batch size of 50 and the Adam optimizer. Our implementations were based on Python (v3.7.5), PyTorch (v1.5.1), and PostgreSQL (v13.0). All experiments were conducted on a machine with Intel(R) Xeon(R) Gold 5122 CPU and 32 GB DRAM. Our source code and query sets are open-source on github[1].

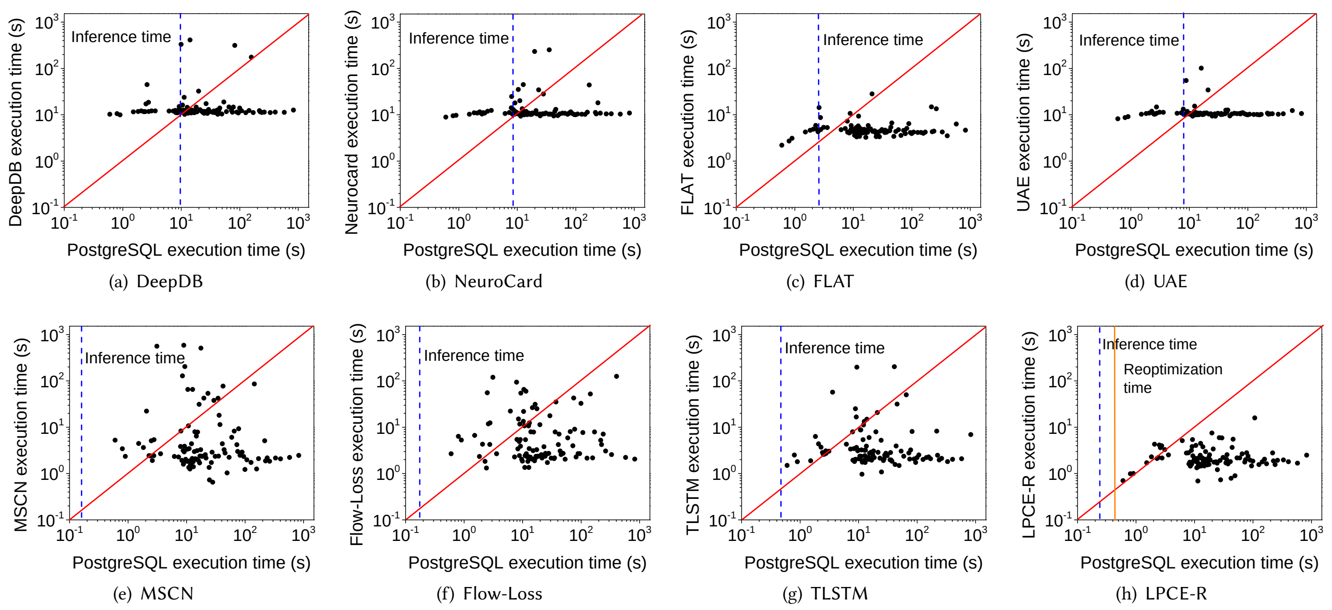
7.2 End-to-End Query Execution Time

For a query, we define the *execution time reduction* (reduction for short) of a learning-based estimator w.r.t. PostgreSQL as

$$R = \frac{T_{Postgres} - T_{Learn}}{T_{Postgres}}, \tag{9}$$

in which $T_{Postgres}$ and T_{Learn} are the end-to-end query execution time of PostgreSQL and the learning-based estimator, respectively. A negative reduction means that the learning-based estimator has longer execution time than PostgreSQL. In Table 2, we report some typical percentiles of the reduction for the learning-based methods, where the 5th percentile and 95th percentile correspond to extreme cases. In Figure 12, we decompose the end-to-end execution time of the queries, where *query execution time* is the time taken by the PostgreSQL engine to execute the selected query plan and constitutes part of the end-to-end execution time (discussed in Section 6). To avoid confusion, we refer to the *end-to-end query execution time* as *end-to-end time* hereafter. For the learning-based

[1]https://anonymous.4open.science/r/LPCE-4BFA/



(a) DeepDB (b) NeuroCard (c) FLAT (d) UAE

(e) MSCN (f) Flow-Loss (g) TLSTM (h) LPCE-R

Figure 13: End-to-end query execution time of the learning-based estimators and PostgreSQL for *Join-eight* queries.

estimators, we also plot the end-to-end time of each *Join-eight* query as scatter plot in Figure 13.[2] A point on the left of the diagonal line indicates that the learning-based estimator has longer end-to-end time than PostgreSQL for a query, and the dotted line is the model inference time. We can the following observations from the results.

1. High estimation accuracy is crucial: Figure 12 shows that all learning-based estimators have significantly shorter aggregate end-to-end time than PostgreSQL, and the reductions are positive for most percentiles in Table 2. These results echo Figure 13, which shows that the learning-based estimators reduce the end-to-end time of most queries as most points are below the diagonal line. For some queries, the reductions are significant, e.g., from 100s-1,000s on PostgreSQL to several seconds. This is because the histogram-based estimator of PostgreSQL has poor accuracy and leads to poor execution plans. Thanks to their good estimation accuracy, the data-driven and hybrid estimators (i.e., DeepDB, NeuroCard, FLAT and UAE) generally spend less time for query execution than the query-driven estimators (i.e., MSCN, Flow-Loss, TLSTM and LPCE-I) in Figure 12. In the same type of learning-based estimators, estimation accuracy is also important. For example, because of its poor estimation accuracy, MSCN has the worst end-to-end time and reduction among the query-driven estimators.

2. Short model inference time matters: Although the data-driven and hybrid estimators have short query execution time in Figure 12, Table 2 shows that they have large negative reductions for end-to-end time at the 5th percentile, which means that they perform significantly worse than PostgreSQL for some queries. Figure 13 shows that this is caused by their long inference time because learning-based estimators cannot improve queries whose end-to-end time on PostgreSQL is shorter than the model inference time (i.e., points on the left of the dotted line). Due to high model inference cost, DeepDB, NeuroCard, and UAE have longer end-to-end time than LPCE-I in Figure 12 although their query execution time are shorter. Model inference cost also explains why all learning-based estimators have worse 5th percentile reduction on *Join-eight* than *Join-six* in Table 2—a query needs up to 127 and 511 cardinality estimations for *Join-six* and *Join-eight*, respectively. Due to the same reason, for the data-driven and hybrid estimators, model inference takes a much larger portion of the end-to-end time for *Join-eight* than *Join-six* in Figure 12. These phenomenons show that for queries with either short execution time or many joins, short model inference time is key to performance.

3. LPCE-R **balances inference time and estimation accuracy:** Table 2 shows that LPCE-R consistently outperforms the baselines across different query sets and percentiles. Compared with the best-performing baseline (i.e., FLAT), LPCE-R takes only 86.5% and 73.8% of its aggregate end-to-end time for *Join-six* and *Join-eight*, respectively. As shown in Figure 12, this is because LPCE-R enjoys both short inference time and high estimation accuracy. On the one hand, the short inference time allows LPCE-R and LPCE-I to achieve small negative reductions at the 5th percentile in Table 2, which resolves the popular concern that a learning-based estimator may significantly degrade the performance for some queries. The re-optimization time is also short because the model for estimation refinement has the same simple structure as the one for initial estimation. On the other hand, the high estimation accuracy enables LPCE-R to find good execution plans, and thus its query execution time is short in Figure 12.

Benefits of query re-optimization: To understand the gain of re-optimization, we compare LPCE-I and LPCE-R for the end-to-end time of queries that trigger re-optimization (39 and 65 for *Join-six* and *Join-eight*, respectively) in Figure 14. The results show that LPCE-R reduces the end-to-end time of LPCE-I by 3.19x and 3.32x

[2]The scatter plots for the *Join-six* queries resemble Figure 13, and thus we show them in the technical report due to space limit.

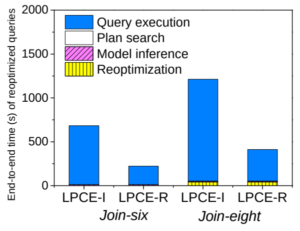


Figure 14: Time decomposition for the re-optimized queries.

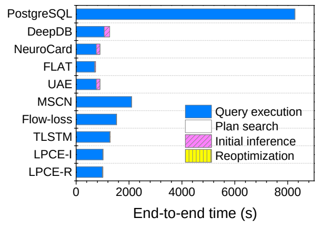


Figure 15: End-to-end query execution time for *Join-three* queries.

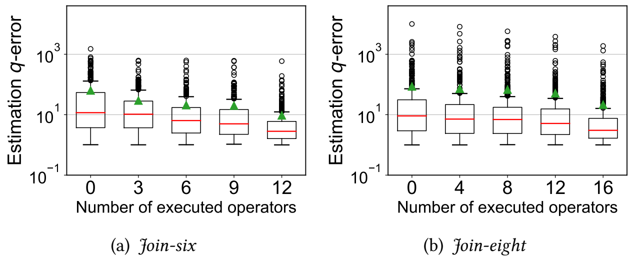


Figure 16: The change of the mean q-error for LPCE-R in the query execution process.

for *Join-six* and *Join-eight*, respectively. Our detailed profiling finds that re-optimization reduces the end-to-end time for most of the queries, especially those having a long running time with LPCE-I. To explain the performance gain of LPCE-R over LPCE-I, Figure 16 shows how the estimation errors of LPCE-R change in the query execution process. The query plan has 14 operators and 18 operators for *Join-six* and *Join-eight*, respectively. The results in Figure 16 shows that LPCE-R effectively reduces the estimation error in the query execution process. For example, when the number of executed operators increases from 3, 6, 9 to 12 on *Join-six*, LPCE-R gradually reduces the mean q-error from 33.5, 22.7, 17.4 to 10.3.

In Figure 17, we provide an example of how the refined cardinality estimations of LPCE-R help find better execution plan[3]. LPCE-I underestimates cardinality for the join result of table T_1, T_2 and T_3, and thus choose nested loop join to join with table T_4, which is expensive for large tables. The large estimation error (3,528 vs. 991,975) triggers re-optimization, and LPCE-R significantly reduces the estimation error. Restarting from the executed sub-plan, hash join is used to join with the results of table T_1, T_2 and T_3, and the left-deep plan generated with LPCE-I is adjusted to a bushy tree. The end-to-end time is reduced by more than 2x.

Limitations of LPCE**:** Although LPCE-R outperforms the baselines in the previous experiments, there can be cases in which the baselines have shorter end-to-end time than LPCE. One such case is when the queries have a small number of joins. For a query joining n relations, plan search in PostgreSQL estimates up to $2^n - 1$ cardinalities. Thus, when n is small, the high inference cost of data-driven and hybrid estimators become less significant, while their high estimation accuracy leads to good execution plans. For such an

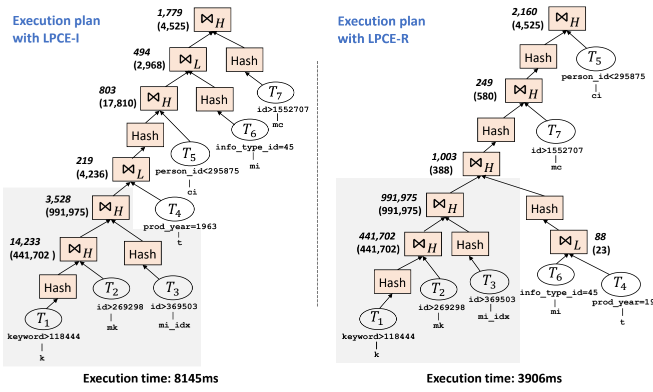


Figure 17: An example of query re-optimization. Upright number for real cardinality and italic number for estimated value. Dashed area indicates executed sub-plan.

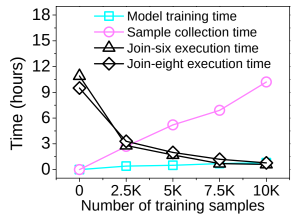


Figure 18: Training and execution time with varying number of samples.

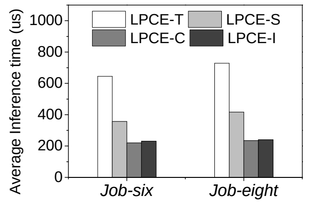


Figure 19: Average model inference time for one cardinality.

example, we report the end-to-end time of 500 queries with 3 joins in Figure 15, where FLAT and NeuroCard outperforms LPCE-R.

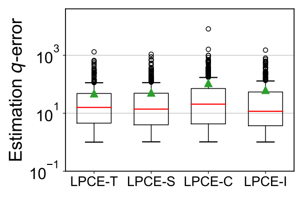
7.3 Design Choices of LPCE

In this part, we evaluate the designs of LPCE-I and LPCE-R. Recall that LPCE-I has three differences from existing query-driven estimators, i.e., the SRU model, model compression via knowledge distillation, and the node-wise loss function.

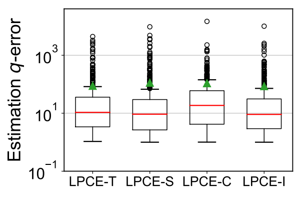
Training dynamics: Figure 18 reports how the sample collection time, model training time and end-to-end query execution time change with the number of training queries. The results show that the execution time decreases monotonically with more training queries for both *Join-six* and *Join-eight* but the return diminishes. However, the sample collection time and model training time increase linearly with sample queries, and the time for sample collection dominates training costs. To reduce the sample collection delay, one practical deployment suggestion is to adopt *progressive training*, where LPCE is deployed after training with a small number of samples and re-trained periodically when more samples (e.g., 2x of the previous training episode) are collected. This strategy not only piggybacks sample collection with normal query execution but also enables LPCE to adapt to changing data and query distributions.

SRU model and knowledge distillation: The SRU model and knowledge distillation contribute to the short inference time of LPCE-I because SRU is lighter than LSTM, and knowledge distillation compresses LPCE-I to smaller size. We check how the two designs affect inference time and estimation accuracy in Figures 19

[3]See https://anonymous.4open.science/r/LPCE-4BFA/LPCE_speedup_example.md for more examples

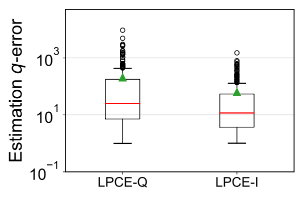


(a) *Join-six*

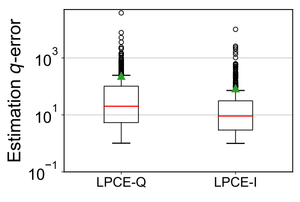


(b) *Join-eight*

Figure 20: Effect of SRU and distillation on accuracy.



(a) *Join-six*



(b) *Join-eight*

Figure 21: Effect of node wise loss function on accuracy.

and 20, respectively. LPCE-T adopts the LSTM model while LPCE-S uses the SRU model, and both LPCE-T and LPCE-S are not compressed. LPCE-C directly trains a model with the same size as LPCE-I (i.e., without knowledge distillation) while LPCE-I uses both SRU and knowledge distillation. The results (i.e., LPCE-T vs. LPCE-S) show that changing the model from LSTM to simpler SRU has almost no influence on accuracy but speeds up inference by 1.7x. LPCE-C and LPCE-I further speed up LPCE-S by 1.8x because they use a smaller model. However, LPCE-C has significantly larger estimation error than LPCE-S while LPCE-I matches the accuracy of the full LPCE-S model. This is because the LPCE-C model is over 10x smaller than LPCE-S and thus has weaker ability to learn. However, knowledge distillation effectively helps LPCE-I to learn by fitting the output of the full LPCE-S model.

Node-wise loss function: Our LPCE-I and LPCE-R are trained with the node-wise loss function while both MSCN and LSTM are trained using the query wise-loss function. We check how the node-wise loss function affects estimation accuracy in Figure 21, in which LPCE-Q shares the structure of LPCE-I but adopts the query-wise loss function. The results show that using the node-wise loss function significantly improves accuracy, which may be explained by two reasons. First, the node-wise loss function enlarges the training set by considering the estimation errors of all sub-plans and can be regarded as a form of data argumentation. Second, the node-wise loss function allows supervision for all internal nodes in an execution plan, which leads to more informative intermediate representations and consequently better accuracy.

Design of LPCE-R**:** Recall that our progressive cardinality refinement model LPCE-R uses a hybrid design with three modules. Module cardinality and content are used to embed the executed operators, and the difference is that cardinality has access to the cardinalities of the executed operators while content does not. The refine module merges the embedding produced by cardinality and content, and processes the remaining operators for the final result. We compare LPCE-R with two alternative designs. (1) LPCE-R-*Single* uses only one module sharing the same structure with cardinality in

Table 3: Percentiles of cardinality estimation q-error for different designs of the progressive model on *Join-eight*.

	Executed Operators	50th	75th	95th	99th	mean
LPCE-R	4	7.15	21.6	66.3	1203	72.6
	8	6.93	18.9	65.8	1877	67.8
	12	5.41	16.2	56.9	1561	51.7
LPCE-R -*Single*	4	17.1	83.1	396	5377	433
	8	12.6	68.9	314	5294	247
	12	8.69	45.1	148	2688	156
LPCE-R -*Two*	4	10.2	37.0	123	3006	123
	8	9.11	33.9	102	2956	88.3
	12	7.16	25.3	97.8	1921	64.8

LPCE-R, and it has access to the cardinalities of intermediate results. The real cardinalities are used for training while during inference the executed operators use the real cardinalities and the remaining operators use the estimated cardinalities. (2) LPCE-R-*Two* uses module cardinality and refine of LPCE-R, in which module cardinality processes the executed operators with real cardinalities while module refine (which does not access cardinalities) takes inputs from module cardinality and processes the remaining operators.

We compare the estimation error of LPCE-R, LPCE-R-*Single* and LPCE-R-*Two* in Table 3. The results show that LPCE-R consistently outperforms LPCE-R-*Single* and LPCE-R-*Two*. LPCE-R-*Single* has the worst performance in most cases because it is trained using the real cardinalities but uses the estimated cardinalities of the remaining operators for inference. Compared with LPCE-R, LPCE-R-*Two* does not use module content, which embeds the contents of the executed operators (e.g., $R.a$<10 and $R.b$>100) without using cardinalities. We conjecture that module cardinality tends to focus on the real cardinalities when embedding the executed operators but the contents of the executed operators (such as a selection on one column) are also important as they may influence the remaining operators. Thus, LPCE-R outperforms LPCE-R-*Two* because module cardinality complements module content by providing more information about the executed operators.

8 CONCLUSION

We propose LPCE, a learning-based cardinality estimator that progressively refines cardinality estimations during query execution process. We observed that to achieve short end-to-end query execution time, both fast inference and high estimation accuracy are crucial but no existing estimators can meet the two requirements. Thus, LPCE adopts a novel framework, which uses a query-driven estimator for fast initial estimation and exploits the executed sub-plans to correct large estimation errors for the remaining operators. We propose techniques including node-wise loss function, knowledge distillation-based model compression, and a refinement model with three modules to instantiate LPCE. Extensive experiments show that LPCE achieves shorter end-to-end query execution time than state-of-the-art learning-based estimators, especially in the worst cases. We think learning-based progressive cardinality estimation is a general idea and can be applied to other estimators (e.g., data-driven and hybrid). To reap the full benefits of progressive estimation, topics such as when to trier query reoptimization and how to tailor the plan search algorithm may be explored.

REFERENCES

[1] Riham Abdel Kader, Peter Boncz, Stefan Manegold, and Maurice Van Keulen. 2009. ROX: run-time optimization of XQueries. In *SIGMOD*. 615–626.

[2] Ashraf Aboulnaga and Surajit Chaudhuri. 1999. Self-tuning Histograms: Building Histograms Without Looking at Data. In *SIGMOD*. 181–192.

[3] Musbah Abdulkarim Musbah Ataya and Musab AM Ali. 2019. Acceptance of Website Security on E-banking. A-Review. In *2019 IEEE 10th Control and System Graduate Research Colloquium (ICSGRC)*. 201–206.

[4] Ron Avnur and Joseph M Hellerstein. 2000. Eddies: Continuously adaptive query processing. In *SIGMOD*. 261–272.

[5] Amol Deshpande, Minos N. Garofalakis, and Rajeev Rastogi. 2001. Independence is Good: Dependency-Based Histogram Synopses for High-Dimensional Data. In *SIGMOD*. 199–210.

[6] Anshuman Dutt and Jayant R Haritsa. 2014. Plan bouquets: query processing without selectivity estimation. In *SIGMOD*. 1039–1050.

[7] Anshuman Dutt, Chi Wang, Vivek Narasayya, and Surajit Chaudhuri. 2020. Efficiently approximating selectivity functions using low overhead regression models. *PVLDB* 13, 12 (2020), 2215–2228.

[8] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. 2019. Selectivity estimation for range predicates using lightweight models. *PVLDB* 12, 9 (2019), 1044–1057.

[9] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, et al. 2021. Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. *Proceedings of the VLDB Endowment* 15, 4 (2021), 1–12.

[10] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. 2020. Deep learning models for selectivity estimation of multi-attribute queries. In *SIGMOD*. 1035–1050.

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

[12] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531* (2015).

[13] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural computation* 9, 8 (1997), 1735–1780.

[14] Navin Kabra and David J DeWitt. 1998. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD*. 106–117.

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

[16] Hai Lan, Zhifeng Bao, and Yuwei Peng. 2021. A Survey on Advancing the DBMS Query Optimizer: Cardinality Estimation, Cost Model, and Plan Enumeration. *Data Science and Engineering* (2021), 1–16.

[17] Tao Lei, Yu Zhang, and Yoav Artzi. 2017. Training RNNs as Fast as CNNs. *arXiv preprint arXiv:1709.02755* (2017).

[18] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9, 3 (2015), 204–215.

[19] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. 2017. Cardinality Estimation Done Right: Index-Based Join Sampling. In *CIDR*.

[20] Jie Liu, Wenqian Dong, Qingqing Zhou, and Dong Li. 2021. Fauce: fast and accurate deep ensembles with uncertainty for cardinality estimation. *Proceedings of the VLDB Endowment* 14, 11 (2021), 1950–1963.

[21] Guy Lohman. 2014. Is query optimization a "solve" problem. In *Proc. Workshop on Database Query Optimization*, Vol. 13. Oregon Graduate Center Comp. Sci. Tech. Rep.

[22] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoia, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2021. Flow-Loss: Learning Cardinality Estimates That Matter. *Proceedings of the VLDB Endowment* 14, 11 (2021).

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

[24] Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, and Miso Cilimdzic. 2004. Robust query processing through progressive optimization. In *SIGMOD*. 659–670.

[25] Thomas Neumann and Cesar Galindo-Legaria. 2013. Taking the edge off cardinality estimation errors using incremental execution. *Datenbanksysteme für Business, Technologie und Web (BTW) 2019* (2013).

[26] Technical report. 2022. Speeding Up End-to-end Query Execution via Learning-based Progressive Cardinality Estimation. (Feb. 2022). https://anonymous.4open.science/r/LPCE-4BFA/techreport.pdf

[27] Danilo Rezende and Shakir Mohamed. 2015. Variational inference with normalizing flows. In *International conference on machine learning*. 1530–1538.

[28] Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. 2015. FitNets: Hints for Thin Deep Nets. In *ICLR*.

[29] Michael Stillger, Guy M Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO-DB2's learning optimizer. In *VLDB*, Vol. 1. 19–28.

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

[31] Ji Sun, Jintao Zhang, Zhaoyan Sun, Guoliang Li, and Nan Tang. 2021. Learned cardinality estimation: A design space exploration and a comparative evaluation. *Proceedings of the VLDB Endowment* 15, 1 (2021), 85–97.

[32] Jiayi Wang, Chengliang Chai, Jiabin Liu, and Guoliang Li. 2021. FACE: a normalizing flow based cardinality estimator. *Proceedings of the VLDB Endowment* 15, 1 (2021), 72–84.

[33] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2021. Are We Ready For Learned Cardinality Estimation? *Proc. VLDB Endow.* 14, 9 (2021), 1640–1654.

[34] Peizhi Wu and Gao Cong. 2021. A Unified Deep Model of Learning from both Data and Queries for Cardinality Estimation. In *SIGMOD*. 2009–2022.

[35] Renzhi Wu, Bolin Ding, Xu Chu, Zhewei Wei, Xiening Dai, Tao Guan, and Jingren Zhou. 2021. Learning to be a Statistician: Learned Estimator for Number of Distinct Values. *Proc. VLDB Endow.* 15, 2 (2021), 272–284.

[36] Wentao Wu, Jeffrey F Naughton, and Harneet Singh. 2016. Sampling-based query re-optimization. In *SIGMOD*. 1721–1736.

[37] Ziniu Wu, Amir Shaikhha, Rong Zhu, Kai Zeng, Yuxing Han, and Jingren Zhou. 2020. BayesCard: Revitilizing Bayesian Frameworks for Cardinality Estimation. *arXiv e-prints* (2020), arXiv–2012.

[38] Ziniu Wu, Rong Zhu, Andreas Pfadler, Yuxing Han, Jiangneng Li, Zhengping Qian, Kai Zeng, and Jingren Zhou. 2020. FSPN: A New Class of Probabilistic Graphical Model. *CoRR* abs/2011.09020 (2020). https://arxiv.org/abs/2011.09020

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

[40] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Peter Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep Unsupervised Cardinality Estimation. *PVLDB* 13, 3 (2019), 279–292.

[41] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Ruslan Salakhutdinov, and Alexander Smola. 2017. Deep sets. *arXiv preprint arXiv:1703.06114* (2017).

[42] Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, and Bin Cui. 2021. FLAT: Fast, Lightweight and Accurate Method for Cardinality Estimation. *Proc. VLDB Endow.* 14, 9 (2021), 1489–1502.

[43] Rong Zhu, Tianjing Zeng, Andreas Pfadler, Wei Chen, Bolin Ding, and Jingren Zhou. 2021. Glue: Adaptively Merging Single Table Cardinality to Estimate Join Query Size. *arXiv preprint arXiv:2112.03458* (2021).

9 APPENDIX

9.1 Blocked Nested Loop Join in PostgreSQL

In Section 6.2, we modify the both side of pipeline processing at nested loop join as blocked processing at outer side, so that the nested loop join can work as a checkpoint.

Query execution at PostgreSQL is in manner of pipeline processing. As shown in Figure 22, the execution of merge-sort join can be partitioned into three pipelines. Two pipelines collect and sort the tuples from outer and inner subplan respectively, and the other one evaluates join qualification. Hash join works in similar way with two pipelines. One pipeline collects the tuples from inner subplan and builds up the hash table, and the another gets the tuples from outer subplan and evaluates join qualification. Hence, hash and nest loop join naturally have at least one pipeline to materialize the intermediate tuples. We can place the checkpoint the end of the pipeline, and detect the estimation error.

However, nest loop join can be executed within one pipeline, as shown in Algorithm 1. The pipeline gets one tuple from outer plan, and then scans the all tuples from inner plan for join evaluation. The pipeline generates join results and pass them to subsequent subplan without materializing any tuples. In order to enable nest loop join to materialize intermediate tuples, we block the processing of outer plan, and nest loop join is thus partitioned into two pipelines in Algorithm 2. One pipeline collects all the tuples from outer plan,

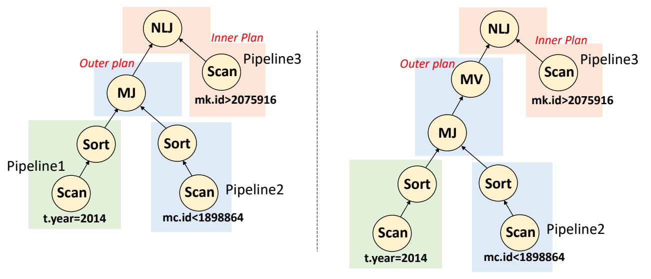


Figure 22: The nested loop join in pipeline processing of PostgreSQL (left), and modified blocked outer side to support re-optimization (right).

and stores them in data structure of PostgreSQL - '*tuplestorestate*'. Then the another gets one from *tuplestorestate* and then scan the all tuples from inner plan for join evaluation. Note that the tuple does not necessarily store in disk, and can cache in memory if there are sufficient space. *tuplestorestate* has the configuration '*allowedMem*' to set the allowed memory space.

Algorithm 1: ExecNestLoop: Nested loop join at PostgreSQL

```
Require: outerPlan, innerPlan
  while 1 do
    outerTupleSlot = ExecProcNode(outerPlan)
    // Get an outer tuple
    if outerTupleSlot is NULL then
      break
    end if
    for (;;) do
      innerTupleSlot = ExecProcNode(innerPlan);
      // Get an inner tuple
      if innerTupleSlot is NULL then
        break
      end if
      ExecQual(innerTupleSlot, outerTupleSlot);
      // Join qualification determine
    end for
  end while
```

Algorithm 2: ExecNestLoop: blocked outer plan of nest loop join at PostgreSQL

```
Require: outerPlan, innerPlan
  while (1) do
    outerTupleSlot = ExecProcNode(outerPlan)
    // Get an outer tuple
    if outerTupleSlot is NULL then
      break
    end if
    Store outerTupleSlot into tuplestore
    // cache outer tuple in tuplestore
  end while
  while tuplestore_gettupleslot() is not NULL do
    outerTupleSlot = tuplestore_gettupleslot()
    for (;;) do
      innerTupleSlot = ExecProcNode(innerPlan);
      // Get an inner tuple
      if innerTupleSlot is NULL then
        break
      end if
      ExecQual(innerTupleSlot, outerTupleSlot);
      // Join qualification determine
    end for
  end while
```