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

Paper ID: xxx

ABSTRACT

Existing learning-based cardinality estimators can be classified into three categories: data-driven, query-driven and hybrid. Data-driven and hybrid estimators enjoy high accuracy but suffer from long inference time. Query-driven estimators have short inference time but incur large estimation errors. However, fast end-to-end query execution requires the estimator to have both short inference time and high estimation accuracy. In this paper, we propose a novel Learning-based Progressive Cardinality Estimator (LPCE) to speed up end-to-end query execution. In particular, LPCE is query-driven, and progressively refines the cardinality estimations during the query execution process, such that the execution plan can be adjusted with more accurate estimates. LPCE consists of two major components: an initial cardinality estimation model and a progressive cardinality refinement model. A suite of tailored designs is incorporated in LPCE to achieve accurate and efficient cardinality estimation. Moreover, we present a re-optimization framework to cooperate LPCE so that it can work in pipeline processing plans. We integrate LPCE into PostgreSQL and conduct extensive experiments on real datasets. The results show that LPCE significantly outperforms existing learning-based estimators in end-to-end query execution time.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability;

KEYWORDS

datasets, neural networks, gaze detection, text tagging

1 INTRODUCTION

Cardinality estimation estimates the size of the intermediate result tables. Query optimizers of database systems rely on the estimated cardinalities to evaluate the costs of the execution plans, and can find the optimal execution plan if the estimations are error-free [21]. Traditional approaches exploit statistics of the tables (e.g., via histogram or sampling) for cardinality estimation [2, 6, 23, 35]. Their errors are large (e.g., many orders of magnitude) as they fail to account for the correlations among the tables [21, 23]. Recently, many learning-based cardinality estimation methods have been proposed [18, 29, 33, 37], which 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 [13, 14, 36] regard cardinality estimation as an unsupervised learning problem and model the joint distribution of the relation tables. Query-driven estimators [9, 18, 29] treat cardinality estimation as a regression problem that maps feature vectors extracted from the 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 the related work.

In this work, we focus on query-driven estimators for three reasons. First, they can be easily integrated into existing database systems in a fashion we call *model-as-a-service*. They require only query samples and result cardinalities, and thus model training and inference can be provided as a service (e.g., on the cloud). The database system and query optimizer are regarded as black boxes, and thus query-driven estimators can be upgraded transparently. This is important given the rapid development of learning-based estimators (e.g., from MSCN [18] in 2019 with about 600 LoC to NeuroCard [36] in 2021 with 8000+ LoC). In addition, their training and inference costs do not have to scale with data size, and hence are more friendly to large-scale data. Second, query-driven estimators do not access the relational tables, and thus are free from data security problems, which are important for areas such as finance and medicine [3]. Third, query-driven estimators typically have shorter inference time than data-driven and hybrid estimators, as we will show shortly in Table 1, and thus add less overhead to end-to-end query execution time.

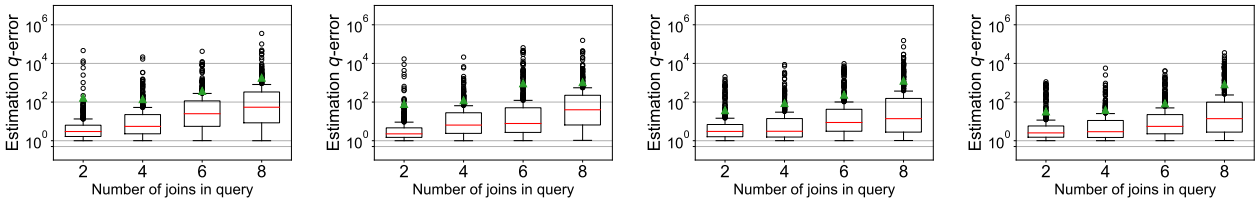
Table 1: A comparison of some learning-based cardinality estimators.

	Estimator	Data access	q-error for estimation	Inference time (ms)
Hybrid	UAE [34]	Yes	5.05	23.4
Data-driven	NeuroCard [36]	Yes	6.39	23.2
	DeepDB [14]	Yes	8.62	29.5
Query-driven	MSCN [18]	No	54.1	0.13
	TLSTM [29]	No	39.8	1.16
	LPCE	No	12.6	0.23

To understand the trade-offs of existing learning-based estimators, we conduct a preliminary test on the IMDB dataset [21]. We generate a set of queries with 8 joins by following [18], and report the estimation error and average inference time of existing learning-based estimators in Table 1. 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 use less information than data-driven and hybrid estimators. To investigate the effect of query complexity (e.g., the number of joins) on existing learning-based estimators, we generate a set of queries whose number of joins ranges from 2 to 8. We plot the estimation error of different estimators in

, ,

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX



(a) MSCN [18] (b) TLSTM [29] (c) NeuroCard [36] (d) UAE [34]

Figure 1: Cardinality estimation accuracy for the queries with different number of joins.

Figure 1. Figures 1(a) and (b) show that the errors of query-driven 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). One reason is that estimation errors propagate and amplify when more operators are involved. Guided by inaccurate estimations, the initial execution plan found by the query optimizer via query-driven estimators can be far from optimal, especially for complex queries.

Inspired by the observations above, we propose a novel query-driven estimator, i.e., *progressive cardinality estimator*, which dynamically refines *cardinality estimations for the remaining operators* in the execution process using *the exact cardinality of the executed operators*, such that the query optimizer can adjust the execution plan to reduce end-to-end execution time. Thus, cardinality estimation enjoys the short inference time of query-driven estimators and high accuracy at the same time. Interestingly, we found that data-driven and hybrid estimators also have larger errors for complex queries, as shown in Figures 1(c) and (d). We plan to extend our progressive cardinality estimation framework to them in the future.

We provide one schematic illustration of progressive cardinality 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., $(R_\sigma \bowtie T_\sigma) \bowtie U_\sigma$, $(R_\sigma \bowtie T_\sigma) \bowtie S_\sigma$, and $(R_\sigma \bowtie T_\sigma) \bowtie S_\sigma \bowtie U_\sigma$). With the refined estimations, the optimizer may find that the plan in Figure 2(b) is better, whose *join order* (for table U_σ and S_σ) and *operator execution method* (from hash join to nested loop join) are different from Figure 2(a). Progressive cardinality estimation poses challenges on both the effectiveness and efficiency of the estimator. Specifically, effectiveness means that the estimator should fully utilize the information of the executed operators to provide more accurate cardinality estimations for the remaining operators. Efficiency means that the estimation model should have a short inference time as the overhead of progressive estimation 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 an 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 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. Modern database systems employ efficient query execution in manner of pipleine processing. We present a framework to use LPCE that place 'checkpoint' at appropriate operations that detect the estimation error and trigger re-optimization when error large, which introduces slight implementation efforts and the overhead worth paying for the re-optimization. A further optimization that brings about more query re-optimization opportunities is proposed for the case that the hardware resources memory is sufficient.

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

- We observe that the estimation errors of query-driven cardinality estimators increase with query complexity and propose progressive cardinality estimation to combat estimation errors and reduce end-to-end query execution time.
- We design the LPCE framework for progressive cardinality estimation, which consists of an initial estimation model (LPCE-I) and a progressive refinement model (LPCE-R). LPCE-I adopts node-wise loss function, SRU-based backbone and knowledge distillation to achieve accuracy and efficiency (Section 4). LPCE-R uses a structure with three modules to effectively extracts information from the executed sub-plans and accurately refines the cardinality estimations for the remaining operators (Section 5).
- We propose a framework to utilize learning-based cardinality estimator in pipeling processing query engines (Section 6) . We integrate LPCE into PostgreSQL, a well-known SQL engine and conduct extensive experiments on the widely used IMDB dataset (Section 8). The results show that LPCE outperforms recent query-driven, data-driven and hybrid estimators in end-to-end query execution time because of its high estimation accuracy, fast inference and progressive estimation refinement.

The remainder of the paper is organized as follows. Section 2 reviews the most relevant works. Section 3 introduces the overall framework and design goals of LPCE. Section 4 presents our initial

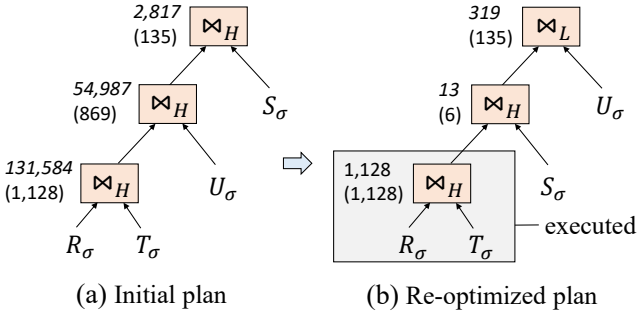


Figure 2: Query re-optimization example, upright number for real cardinality and italic number for estimated value. $\bowtie_L$ indicates nested loop join and $\bowtie_H$ is hash join.

cardinality estimation model LPCE-I. Section 5 discusses the design of our progressive cardinality refinement model LPCE-R. Section 6 describes how we utilize LPCE in pipelined query execution and illustrate the example of PostgreSQL. Section 8 reports our extensive experimental evaluation. Section 9 draws the concluding remarks.

2 RELATED WORK

In this part, we review works that are most relevant to ours.

Traditional cardinality estimators: *Histogram-based* cardinality estimation methods [2, 6, 12] have been widely used in many industrial database systems as they are simple and have very low overhead. However, they lack the ability to capture the data correlations among the tables as they make the attribute-value-independence assumption. *Sampling-based* approaches [5, 10, 32] outperform histogram-based methods as the correlations in data are naturally captured by data samples. However, sampling-based approaches have two limitations: (i) empty sampling set of join result; and (ii) high sampling overhead. Recent works [22, 38] have proposed index-based and materialized sampling strategies to alleviate the problem of empty result. However, it is still difficult to use sampling to evaluate the cost of all execution plans as it incurs high space- and time- costs.

Learning-based cardinality estimators: Recently, the database community recognized the potential of replacing traditional cardinality estimation methods by learning-based models (e.g., neural network, autoregressive model) [19, 33]. Existing learning-based estimators can be classified into three categories: *query-driven*, *data-driven* and *hybrid*.

Data-driven cardinality estimators [14, 36] share the same idea with traditional cardinality estimation methods, i.e., they hope to capture the correlations and distributions of data across the tables. DeepDB [14] adopts relational sum product networks (RSPN) to capture the probability distribution among relations, and translates a query into the evaluations of probabilities and expectations based on RSPN. Naru [37] adopts autoregressive models, i.e., masked autoencoder [11] and transformer [31] to estimate the selectivity of equal and range predicates. NeuroCard [36] trains a single deep autoregressive model based on samples collected from the full outer join result of all relations. The model can be used to answer complex queries with joins on any subset of the relations.

Query-driven cardinality estimators [9, 13, 18, 28, 29] formulate cardinality estimation as a regression problem. The contents of queries and their true cardinalities are used as training data to learn a mapping from queries to cardinalities. [8] uses regression techniques such as XGBoost, to train the model to produce approximate cardinality labels. MSCN transforms a query into a feature vector and uses a multi-set convolution network [39] to map it to cardinality estimation. TLSTM processes a query execution plan recursively following its tree structure using LSTM (a kind of RNN) [16].

Hybrid cardinality estimator UAE [34] learns the joint data distribution among the tables as in the data-driven estimators, and uses query samples as auxiliary information at the same time. With a unified deep autoregressive model, UAE learns from data in an unsupervised manner and query samples in a supervised manner. As we have discussed in Section 1, existing cardinality estimators cannot achieve *high accuracy* and *fast inference* simultaneously, which are necessary for short end-to-end query execution time. ~~LPCE is general framework, which enjoys the fast inference of query-driven estimators and progressively refines estimations to correct large errors. We have shown that data-driven and hybrid estimators also have larger errors for complex queries, and thus may also benefit from our progressive cardinality estimation methodology.~~

Query re-optimization: Although extensive efforts have been made to improve the accuracy of cardinality estimation, the errors can still be large for complex queries, e.g., those with multiple joins. Query re-optimization techniques [1, 4, 7, 17, 26, 35] have been proposed to combat the influence of large estimation errors on query optimization. The general idea is that the query optimizers first choose an initial execution plan, and then exploit extra knowledge to adjust to a better execution plan. There are two classes of query re-optimization techniques. (1) *Re-optimizing during query execution.* For example, ROX [1] samples data from the intermediate results, and tries different operator implementations and join orders on the samples to adjust the remaining execution plan. (2) *Re-optimizing before query execution.* For example, Wu et al. [35] use a sampling-based method to detect estimation errors and then adjust the execution plan before running it. Our LPCE differs from these query re-optimization techniques in that they are sampling-based while LPCE uses machine learning to exploit information in the executed sub-queries for estimation refinement.

3 OVERVIEW OF THE LPCE FRAMEWORK

We consider *select-project-equijoin-aggregate* [18, 24] queries in the following form:

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

where R, U, S and T are relational tables, σ represents the filtering predicates (e.g., R.a < 10, which returns tuples in R whose attribute a is smaller than 10) and the filtering operator can be $<$, $\leq$, $=$, $>$ and $\geq$. Complex predicates such as "IN, LIKE, NOT LIKE" is also supported as following [29]. For a query, query optimizers find an efficient execution plan (usually expressed as a join tree) by enumerating feasible plans using algorithms such as dynamic programming [21]. The cost of a plan is calculated based on cardinality

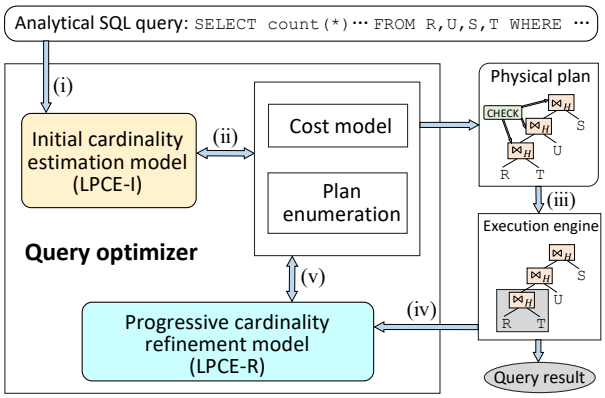


Figure 3: Using LPCE for query end-to-end execution.

estimations of its sub-plans (e.g., $R_\sigma \bowtie T_\sigma$ and $(R_\sigma \bowtie T_\sigma) \bowtie U_\sigma$ for the plan in Figure 2(a)). We focus on query-driven cardinality estimators and design LPCE to combat the large estimation errors for complex queries.

3.1 End-to-end 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 execution process of a query.

i. The query is sent to LPCE-I for initial cardinality estimations of all possible sub-plans when it is submitted;
ii. Using the initial estimations, the query optimizer chooses a good execution plan using its plan search algorithm;
iii. The chosen plan is executed, and the operators chosen as checkpoints (CHECK) monitor the estimation error if the actual cardinalities differ significantly from the initial estimation;
iv. For the feasible sub-plans of the remaining operators, LPCE-R is invoked to refine their cardinality estimations by exploiting the actual cardinalities of the executed sub-plans;
v. Based on the refined estimations, the query optimizer adjusts execution plan of the remaining operators for better efficiency.

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 (w.r.t. the initial estimations). LPCE is most beneficial for 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 reduce the end-to-end execution time of queries, LPCE needs to meet three key goals, i.e., *high accuracy, progressive refinement and fast inference*, which we elaborate as follows.

High Accuracy: Theoretically, query optimizers can find the optimal execution plan if the cardinality estimations and cost model are exact [21]. However, the errors of existing query-driven cardinality estimators can reach several orders of magnitude [21, 23]. To guide query optimizers to a good initial plan, LPCE-I should provide high accuracy for initial cardinality estimations. Accurate initial estimation also reduces the frequency of estimation refinement and query re-optimization, which incur extra overhead. For this goal, we design the node-wise loss function in Section 4.

Fast inference: Learning-based cardinality estimators are shown to significantly outperform traditional ones (e.g., histogram-based, sampling-based) in accuracy [13, 14, 18, 29, 36, 37]. However, a key concern of their applications in real database systems is the long inference time, which may outweigh their advantages in accuracy. Thus, both the initial estimation and progressive refinement models of LPCE should support fast inference. To achieve this goal, we use a light-weight model backbone and further compress it via knowledge distillation in Section 4.

Many other problems are also important for learning-based cardinality estimators, e.g., handling data updates, reducing the number of required samples and shortening training time. We do not consider them in our design goals and leave them for future work.

Progressive refinement: LPCE-R should effectively reduce estimation error when more operators are executed such that re-optimization can find good execution plans. For this purpose, the refinement model 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. To use learning-based estimator in piplined query execution, we present a framework to materialize the intermediate results at appropriate operators and trigger the re-optimization with information of done sub-plans..

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.

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) [9, 13, 18, 29, 36]. 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 [18] 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

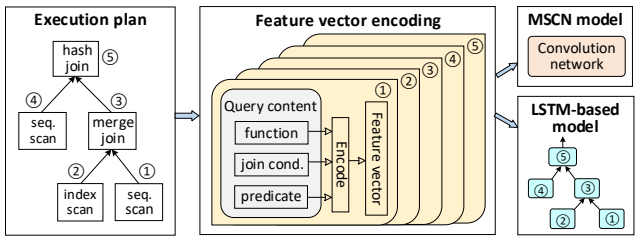


Figure 4: Workflow of learning-based estimation model.

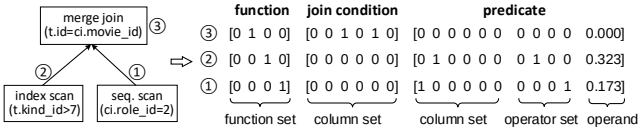


Figure 5: Example for feature encoding.

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 [18, 29], 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 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 [18, 29] for more details about feature encoding.

Model learning: MSCN [18] and TLSTM [29] are two state-of-the-art models for query-driven cardinality estimation with different structures. MSCN stacks the feature vectors of all nodes in an execution plan and uses a multi-set convolution network [39] to map them to cardinality estimation. TLSTM processes an execution plan recursively following its tree structure using LSTM [16]. 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.

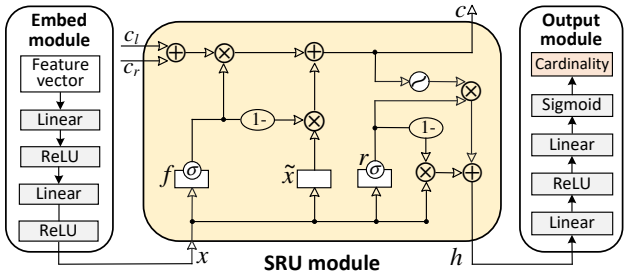


Figure 6: The structure of the SRU-based model in LPCE-I.

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) [20] 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 2.1x 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 function is significant for complex execution plans with a deep tree structure (Section 8).

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* [15, 27], 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

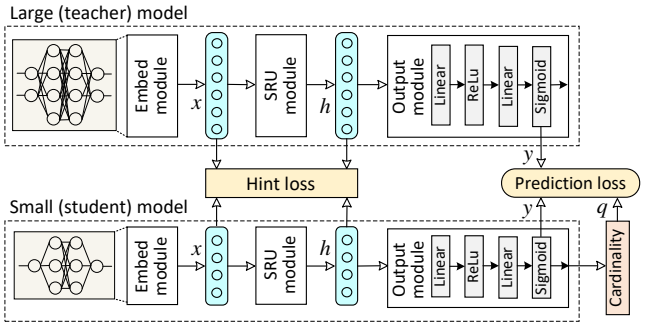


Figure 7: Model compression via knowledge distillation.

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 j^{th} operator in the i^{th} 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.

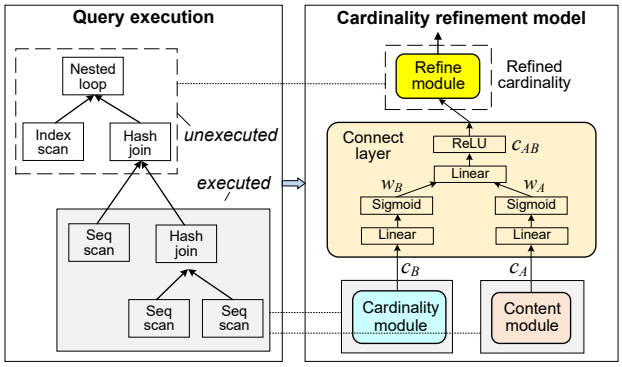


Figure 8: An overview of LPCE-R for estimation refinement.

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

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

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.

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.

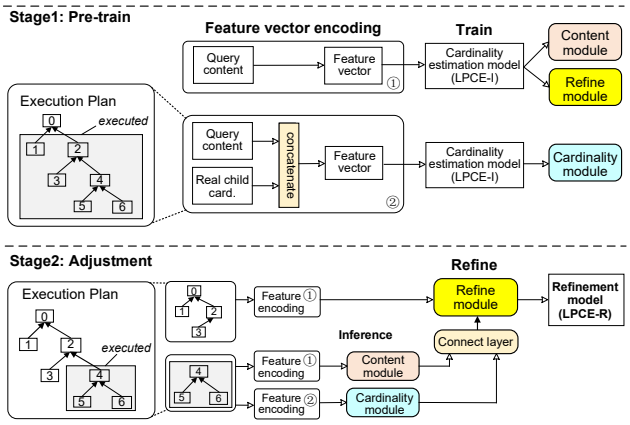


Figure 9: The training workflow of LPCE-R.

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 section, we present how to use LPCE in a database engine. We first present how to exploit LPCE-I by query optimizer to find a execution plan *before* runtime. We then explain how to implement LPCE-R to support query re-optimization in a pipelined execution plan *during* runtime. We illustrate the usage of LPCE in PostgreSQL, which is also generic to other database engines such as MySQL.

6.1 Query Initial Optimization with LPCE-I

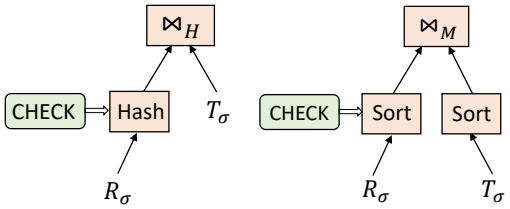
Given a query, the cardinality estimator estimates the cardinalities inside of all possible execution plans to identify the one with minimum cost. To contribute more accurate estimation, we replace the histogram-based cardinality estimator with LPCE-I.

The query optimizer of PostgreSQL uses dynamic programming to enumerate all possible execution plans. 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., $R_\sigma \bowtie T_\sigma \bowtie U_\sigma$). To find the best execution plan for a sub-query at level-i, the optimizer decomposes it into smaller sub-queries at lower levels (called children) and requires cardinality estimations of all children. For example, optimizing $R_\sigma \bowtie T_\sigma \bowtie U_\sigma$ requires cardinality estimations of R_σ, T_σ, U_σ, $R_\sigma \bowtie T_\sigma$, $T_\sigma \bowtie U_\sigma$ and $R_\sigma \bowtie U_\sigma$. Thus, a query that joins n relations requires up to $2^n - 1$ cardinality estimations (when all relations can join with each other).

LPCE-I provides all cardinality estimations. Following [29], a *memory pool* is used to store the cardinality estimations and execution costs of sub-queries. We batch the inference for all sub-queries on the same level as they have the same number of inputs and the feature vectors of a sub-query are small. 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. For 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 queries with a short execution time T_E, T_I can easily dominate T_{end}, in which case learning-based estimators may have longer end-to-end time than PostgreSQL. In the experiments in Section 8, we show that LPCE-I outperforms PostgreSQL in most cases due to its short inference time.



(a) Hash join (b) Sort merge join

Figure 10: LPCE-R **places the checkpoint at hash join (a) and sort merge join (b).**

6.2 Query Re-optimization LPCE-R

PostgreSQL does not provide native support for query re-optimization, and work in manner of pipeline processing. Several operations might be grouped as a pipeline, in which one tuple produced by one operation is passed to next operations that are evaluating simultaneously. However, the query re-optimization expects the materialized temporary relation of all intermediate results when detect large errors, so that the woke done by executed part can be reused, and guarantee duplicated tuples return.

To use LPCE-R in a pipelined execution plan, we follow the *checkpoint* scheme in [25]. We use checkpoint to assert whether the real cardinality of the operator is significantly different from the (initially) estimated cardinality. The re-optimization is triggered to adjust the execution plan once error large at checkpoint. We place the checkpoint at each operation that triggers the materialization of intermediate results. In PostgreSQL, there are some operations that block the processing and accumulate the results in a buffer, and write to disk when the buffer is full. Such operations can be naturally used as checkpoint, such as *sort* and *materialization*. While such operations might not be appeared in plans, join operations are common in SPJ queries. Join implementations in PostgreSQL include hash join, merge join, and nested loop join. As shown in Figure 10, checkpoint (CHECK) can be placed at the inner side of building hash table in hash join, and the both side of sorting node in merge join. For nested loop, while both sides are pipelined by default, we block and place checkpoint at the outer side of scanning small table as shown in Figure 11.

Modifying the outer side of nested loop join as blocked processing would lead to the extra latency and memory usage. We observe the overhead is limited since the nested loop join is chosen only when the cardinality of the outer side is small. Figure 12 shows the query execution time and memory computation for benchmarks IMDB [21] and TPC-H [30]. PostgreSQL-Re is PostgreSQL with blocked nested loop join to support re-optimization. The test queries of IMDB includes 100 queries, each of which triggers at

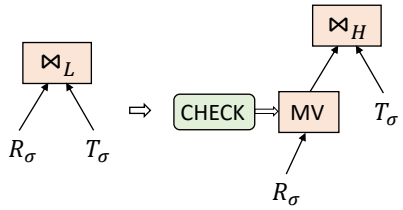
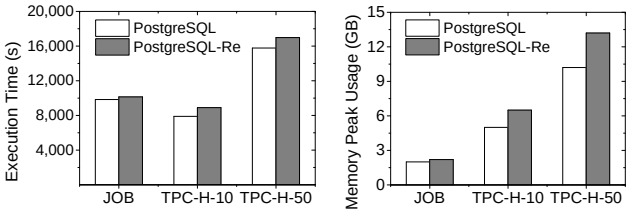


Figure 11: The nested loop implementation at PostgreSQL (a) and LPCE-R places the checkpoint at outer side (b).



(a) Execution time (b) Memory peak consumption

Figure 12: The extra query execution time and memory usage when modifying the nested loop join in PostgreSQL. (data fake)

lease one nested loop join. TPC-H-10 and TPC-H-50 are TPC dataset of the scale factor 10 and 50, respectively. The total query execution slightly prolong by xx% with consuming xx% memory.

We use the q-error at each checkpoint to measure the difference between the actual cardinalities and initially estimated cardinalities, and continue executing the original plan if the q-error is below a threshold (empirically set as 50). Otherwise, re-optimization is triggered at the checkpoint. The intermediate results at the checkpoint remains in buffer (or disk when it exceeds the buffer size), which can be used by subsequent operators. During re-optimization, LPCE-R is invoked to provide refined cardinality estimations, and the optimizer searches the optimal execution plan among those that continues from the checkpoint triggered re-optimization and those that restarts query processing from scratch. The end-to-end execution time of a query that triggers re-optimization 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. Therefore, re-optimization should be triggered when its execution time reduction outweighs T_R. Currently, we use a simple threshold, which may invoke unnecessary re-optimizations or miss re-optimization opportunities. The experiments in Section 8 show that re-optimization with this simple rule already brings benefits and we leave designing more sophisticated policies to trigger re-optimization as future work. From Equation (8), we also observe that the short inference time of LPCE-R is crucial for enjoying the benefits of re-optimization.

6.3 More Opportunities (??)

The scheme in Section 6.2 leads to slight implementation efforts (i.e., modifying the nested loop join as blocked operator) and negligible performance overhead. However, it limits the opportunities of reusing the work of executed subplan. We observe that the hash join are generally used more often in execution plans involving many joins. For example, the xx% join are hash join for 100 queries 6-joins of IMDB. This is because while nested loop is chosen when

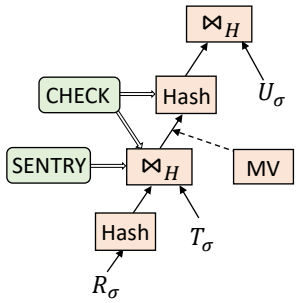


Figure 13: Re-optimization with point to signal materializing the incoming tuple of hash table building side.

one relation is at small size, and sort-merge prefers the relations with sequential characteristic, hash join is more general for common cases. The checkpoints thus often trigger re-optimization at hash join operators. However, the checkpoint at hash join is indeed the materialization of hash table, rather than intermediate tuples. This limits the done work of hash join can only be re-used by hash join at re-optimization plan, while other operators such as nested loop and sort operators miss the opportunity of using done work.

Modern commercial severs often have large memory space as the memory kits have become cheaper. This leads to the acceptable cost of materializing the intermediate tuples since there are enough space to cache them in memory data structure rather then disk (e.g., tuplestore within in PostgreSQL). We thus can trigger the materialization once the checkpoint at hash join detect large error and triggers the re-optimization. To achieve this, we introduce the sentry point. Each sentry point is indeed a checkpoint and works for signalling the next checkpoint to start materialization once the q-error is close to the threshold trigger re-optimization. As shown in Figure 13, we let each checkpoint also works as sentry point. Once q-error (empirically set as 30) at sentry point is close to the threshold of re-optimization (50), we start to collect the result tuples and store in materialization at the next checkpoint if it's hash join.

The trade-off of sentry scheme is the extra cost of materialization and more opportunities of triggering the better execution plan. The end-to-end execution time of a query that triggers re-optimization can be expressed as

$$T_{end} = T_P + T_I + T_R + T_E + T_{MV}, \tag{9}$$

where T_{MV} is the extra materialization at hash join side and other components same in Equation 8. Our experiments in Section 8 show that sentry shows high potential of re-optimizing plan when we have sufficient resources of memory space.

7 DISCUSSION

In this section, we discuss practical 'tips' to shorten preparatory work of LPCE.

LPCE as learning-based estimator need efforts to train the model at offline stage. Building up LPCE includes two stages: query sample collection and model training. LPCE learns from the query samples, and the generalize better with more samples. The samples can be collected from historical execution log and random generation based the dataset as discussed in Section 4.1. We recommend the latter in practical, as it is feasible to provide samples similar to the target query domain. To speed up the sample collection, we can

start model training with a small number of samples and continue collecting samples by executing queries using the model. The model is re-trained progressively (e.g., by 2x more samples each time) until accuracy plateaus. Due to the essential SRU structure of LPCE, the computation of training LPCE is intensive matrix multiplication, modern hardware such GPU and multi-core CPU can be used to accelerate model training stage.

8 EXPERIMENTAL EVALUATION

In this section, we present our empirical studies. Section 8.1 introduces the experiment settings. Section 8.2 evaluates the end-to-end query execution time of LPCE and compares with state-of-the-art solutions. Section 8.3 validates the effectiveness of the key designs (e.g., node-wise loss function, model compression) of LPCE.

8.1 Experiment Settings

Workloads: We used the IMDB dataset [21] 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 [18, 19, 22, 29, 36, 37] as a challenging benchmark for cardinality estimation due to its non-uniform data distributions and complex correlations among the tables. We generate the training and testing queries according to the relational graph for the tables in IMDB following [18]. The training set contains 10,000 sample queries with 6-8 joins, and 10% of these queries are randomly selected as validation. We used two query sets for performance testing: (1) *Join-six*, 500 queries with 6 joins; and (2) *Join-eight*, 500 queries with 8 joins. As shown in Figure 14, 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).

Baselines: We compared our LPCE with the following state-of-the-art learning-based cardinality estimators.

- 2 query-driven estimators, i.e., MSCN [18] and TLSTM [29]. MSCN uses a multi-set convolution network to map the query feature vector to cardinality value while TLSTM uses an LSTM (a kind of recurrent neural network) to estimate the cardinality of a query according to its execution plan.
- 2 data-driven estimators, i.e., DeepDB [14] and NeuroCard [36]. DeepDB adopts the *sum product networks* (SPN) to capture the joint distribution of the relations and assumes conditional independence across the relations. NeuroCard removes the independence assumption and builds a single deep autoregressive (AR) model over all relations.
- 1 hybrid estimator, UAE [34], which learns from data in an unsupervised manner and query samples in a supervised manner.

Our methods include LPCE-I, which only conducts cardinality estimation before query execution (the same as existing learning-based estimators), and LPCE-R, which uses LPCE-I for initial estimation and may trigger query re-optimization and progressive estimation

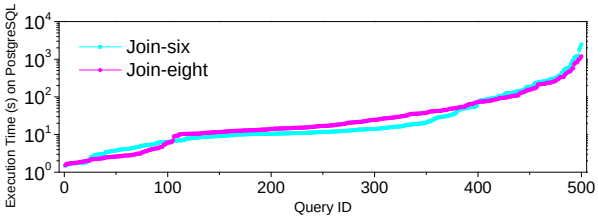


Figure 14: Execution time of the test queries on PostgreSQL. Queries are ordered in the ascending order of the execution time.

refinement if the errors are found to be large. We use end-to-end query execution time as the main performance metric.

Implementation details: For MSCN[1], DeepDB[2] and NeuroCard[3], we used their open-source implementations and adopted the hyper-parameters recommended by their authors. For UAE, we obtained the implementation from its authors. As TLSTM is not open-source, we implemented it on our own and tuned the hyper-parameters to reproduce the results reported by in its paper. Since MSCN and DeepDB do not support range queries on categorical string columns, we encode these columns into integers using dictionary encoding. As described in Section **??**, we replace the histogram-based cardinality estimator of PostgreSQL with the learning-based estimators for performance evaluation.

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, cardinality refinement is triggered when the q-error between the actual cardinality of an intermediate result table and the initial estimation is above 50. 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 512 GB DRAM. Our source code and query sets are open-source on github[4].

8.2 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

$$R = \frac{T_{Postgres} - T_{Learn}}{T_{Postgres}}, \tag{10}$$

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, and a large reduction indicates a significant speedup (e.g., a reduction of 99% means a 100x speedup over PostgreSQL). In Table 2, we report some percentiles of the reduction for the learning-based methods, in which the 5th percentile and 95th percentile correspond to worst and best cases.

[1]https://github.com/andreaskipf/learnedcardinalities/
[2]https://github.com/DataManagementLab/deepdb-public
[3]https://github.com/neurocard/neurocard
[4]https://anonymous.4open.science/r/LPCE-B90D/

We also plot the end-to-end execution time for the learning-based estimators as scatter plot in Figure 15. A point on the left of the diagonal line indicates the learning-based estimator has longer execution time than PostgreSQL, and the dotted line indicates the model inference time. The scatter plots for the *Join-six* queries resemble Figure 15 and we omit them due to space limit. We make 3 observations from Table 2 and Figure 15.

1. Short model inference time is crucial: Table 2 shows that data-driven and hybrid estimators (i.e., DeepDB, NeuroCard and UAE) have poor performance at 5th percentile (also 25th percentile for *Join-eight*), and can significantly prolong the execution time of PostgreSQL for some queries. Figure 15 shows that this can be explained by their long inference time, which is several to tens of seconds and they cannot speed up queries whose execution time on PostgreSQL is shorter than their inference time (i.e., queries on the left of the dotted line in Figure 15). The cost of model inference also explains why data-driven and hybrid estimators have worse 5th percentile on *Join-eight* than *Join-six*—a query needs up to 127 and 511 cardinality estimations for *Join-six* and *Join-eight*, respectively. In contrast, the query-driven estimators (i.e., MSCN, LSTM and LPCE-I) generally performs better at 5th percentile because of their short inference time, which is typically below 1 second.

2. High estimation accuracy matters: Table 2 also shows that the learning-based estimators speed up PostgreSQL in most cases. For example, LPCE achieves a reduction of 99.7% at 95th percentile for *Join-six* queries, which corresponds to a speedup of about 330x. We also observed that learning-based estimators can reduce the execution time from around 1,000s for PostgreSQL to several seconds. This is because the histogram-based estimator of PostgreSQL has poor accuracy and thus leads to poor execution plans. Among the same type of learning-based estimators, estimation accuracy is also important to performance. Although DeepDB, NeuroCard and UAE have similar inference time according to Figure 15, UAE has the best reduction in Table 2 because it jointly utilizes data-based training and query-based training to achieve high accuracy. In contrast, DeepDB has the worst reduction in Table 2 due to the poor accuracy caused by its independence assumptions. In the query-driven estimators, MSCN has significantly shorter inference time than LSTM and LPCE-I, but its reduction is much worse than LSTM and LPCE-I because of poor estimation accuracy.

3. LPCE **balances inference time and estimation accuracy:** Both Table 2 and Figure 15 show that LPCE consistently outperforms the baselines across different query sets and percentiles. The 5th percentiles of LPCE are small (below 10%), which resolves the popular concern that a learning-based estimator may significantly degrades performance for some queries. For more optimistic cases (e.g., 75th or 95th percentile), LPCE achieves larger or comparable speedup over PostgreSQL when compared with other learning-based estimators. LPCE has good performance because it balances inference time and estimation accuracy, which we show in Figure 16 by decomposing the end-to-end execution time of the queries. The results show that for methods with poor cardinality estimation accuracy (e.g., PostgreSQL and MSCN), the query execution time is long as bad execution plans are used. For methods with a long inference time (e.g., UAE and NeuroCard), although query execution time is short, model inference could dominate end-to-end query execution. LPCE-R benefits from using LPCE-I for an efficient and accurate initial estimation, and progressively refining the estimations during re-optimization. Although the re-optimization time of LPCE-R is long as reported in Figure 15, Figure 16 shows that the re-optimization time is negligible in end-to-end time. This is because re-optimization time is only triggered for a few queries with large estimation errors, which we will show shortly.

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

Join-six	5th	25th	50th	75th	95th
DeepDB	-226%	7.85%	56.3%	88.5%	98.1%
NeuroCard	-115%	11.9%	45.9%	71.8%	97.4%
UAE	-116%	44.7%	64.4%	89.7%	98.9%
MSCN	-240%	37.6%	70.6%	91.1%	99.4%
TLSTM	-118%	40.3%	72.9%	92.3%	98.0%
LPCE-I	-11.23%	69.2%	85.4%	94.1%	99.6%
LPCE-R	**-3.01%**	**71.2%**	**86.8%**	**96.5%**	**99.7%**
Join-eight	5th	25th	50th	75th	95th
DeepDB	-445%	-22.5%	22.8%	71.7%	95.5%
NeuroCard	-370%	-14.3%	29.6%	74.5%	95.7%
UAE	-360%	-8.85%	35.2%	75.8%	95.8%
MSCN	-978%	22.2%	72.5%	82.3%	98.1%
TLSTM	-389%	34.3%	75.8%	92.1%	98.6%
LPCE-I	-28.9%	73.0%	86.7%	95.9%	99.3%
LPCE-R	**-9.56%**	**74.5%**	**87.0%**	**96.2%**	**99.3%**

Advantages of progressive estimation refinement: Among the 500 test queries, 49 and 27 queries incur progressively estimation refinement for *Join-six* and *Join-eight*, respectively. To show the gain of progressive refinement, we compare the execution time of these queries with LPCE-I and LPCE-R in Figure 17. The results show that LPCE-R speeds up most of the queries and the speedup is the most significant for queries whose execution time is long with LPCE-I. This is because LPCE-R can correct the significant estimation errors of LPCE-I, which leads to bad execution plans. For a few queries, LPCE-R has longer end-to-end execution time than LPCE-I but the performance degradation is small. This is because LPCE-R still has large errors on some difficult queries and re-optimization comes with overhead. In Figure 20, we report the time decomposition for the re-optimized queries. The results show that LPCE-R significantly reduces the overall execution time of the re-optimized queries and the re-optimization overhead is small. The re-optimization time includes model inference and materialization of intermediate results by checkpoint. We observed that materialization cost typically dominates the re-optimization time and is determined by the size of the intermediate results.

To explain the performance gain of LPCE-R over LPCE-I, Figure 18 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 18 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. One interesting observation is that there are some difficult queries whose estimation errors are large even when many operators are

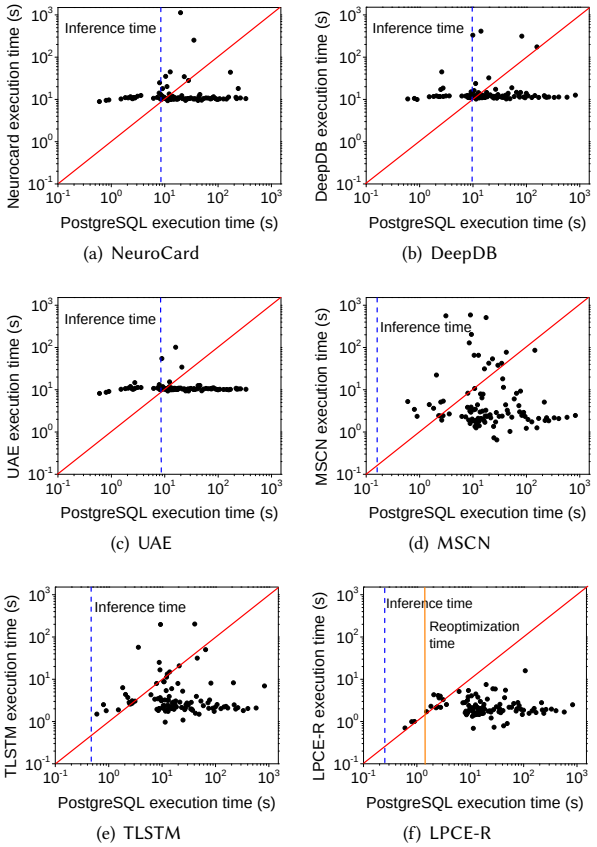


Figure 15: End-to-end execution time of the learning-based estimators and PostgreSQL for *Join-eight* queries.

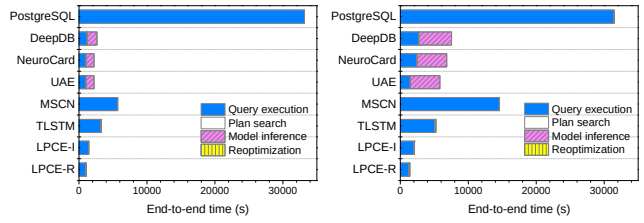


Figure 16: Decomposition of end-to-end query execution time, statistics are aggregated over the 500 test queries.

executed. We conjecture that this is because these queries do not having matching samples in the training set and thus the model cannot generalize to them.

Performance of sentry **re-optimization:** TO-DO...

Limitations of LPCE**:** Although LPCE-R outperforms the baselines in the previous experiments, there can be cases in which the baselines have short end-to-end execution 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 needs to estimate up to $2^n - 1$ cardinalities. Thus, when n is small, the high inference overhead of data-driven and hybrid estimators becomes

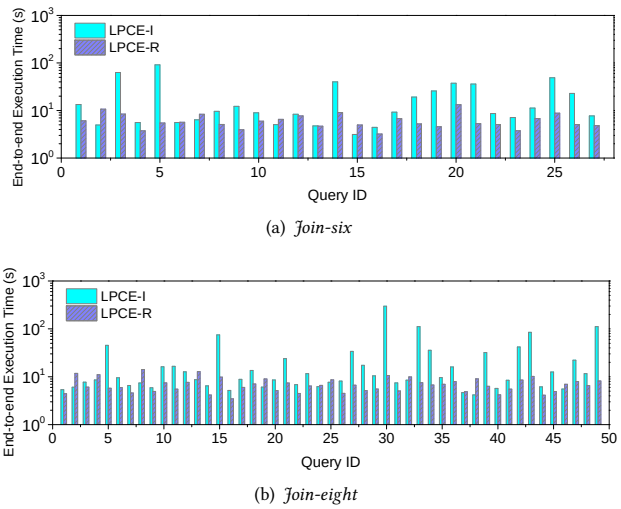


Figure 17: Re-optimization of LPCE-R.

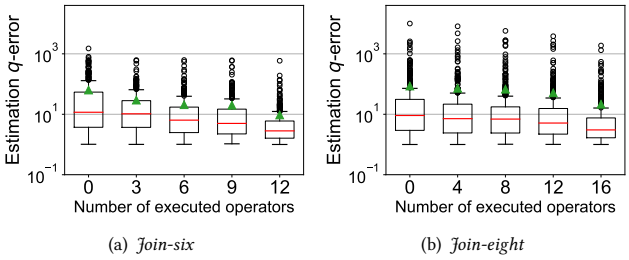


Figure 18: The change of the mean q-error for LPCE-R **in the query execution process.**

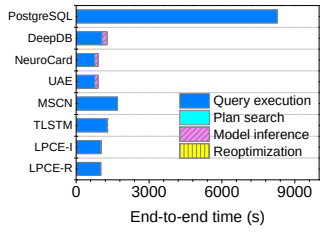


Figure 19: End-to-end execution time for 3-join queries.

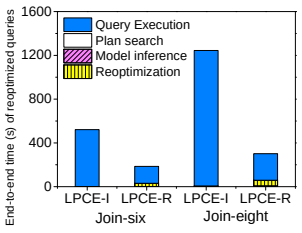


Figure 20: Time decomposition for the re-optimized queries.

less significant. In addition, data-driven and hybrid estimators typically have better estimation accuracy than query-driven estimators. To provide such an example, we report the end-to-end execution time of 500 queries with 3 joins in Figure 19. The results show that NeuroCard and UAE perform the best among the estimators, and the primary reason is that model inference overhead is lower compared with more complex queries.

Impact of query sample size: LPCE learns the knowledge from query samples, and more samples make LPCE generalize better on wider query domain. Figure 21 shows the impact of samples on training time and query execution time. Note that the case with 0 training samples corresponds to vanilla PostgreSQL. When using

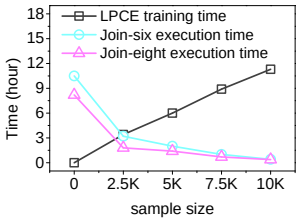


Figure 21: Training time and query execution time with varying number of samples. data fake

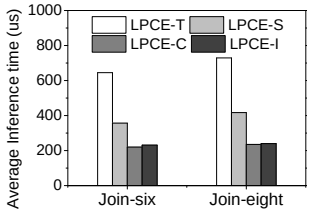


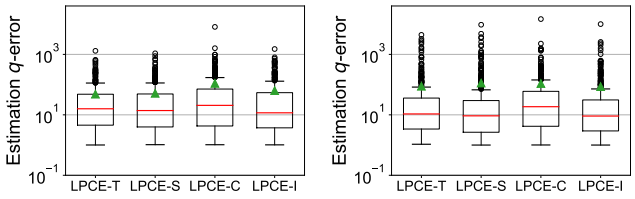
Figure 22: Average model inference time for one cardinality.

more training samples, training time increases but query execution time drops due to better model accuracy.

8.3 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, which we test at first.

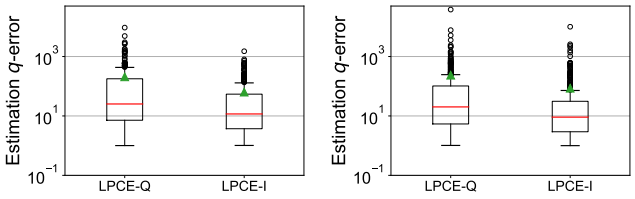
SRU model and knowledge distillation: The SRU model and knowledge distillation contribute to the short inference time of LPCE-I because SRU lighter than LSTM, and knowledge distillation compresses LPCE-I to smaller size. We check how the two designs affect inference time and estimation accuracy in Figures 22 and 23, 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. This is because the SRU model uses fewer matrices and thus reduces the model size to 17.9 MB from 37.5 MB for LSTM. With a model of only 1.5 MB, 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.



(a) *Join-six* (b) *Join-eight*

Figure 23: Effect of SRU and knowledge distillation on estimation accuracy.

Node-wise loss function: Recall that our LPCE-I and LPCE-R are trained with the node-wise loss function while both MSCN and



(a) *Join-six* (b) *Join-eight*

Figure 24: Effect of node wise loss function on estimation accuracy.

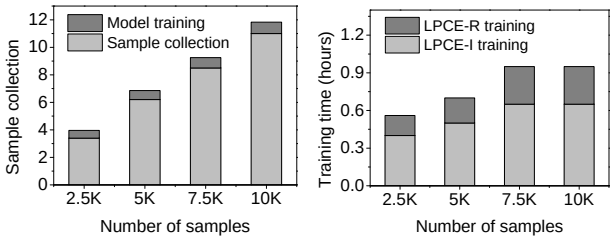
LSTM are trained using the query wise-loss function. We check how the node-wise loss function affects estimation accuracy in Figure 24, in which LPCE-Q shares the structure of LPCE-I but adopts the query-wise loss function in Equation 2. The results show that using the node-wise loss function significantly improves accuracy, which may be explained from two aspects. 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 final 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 their difference is that the cardinality module has access to the cardinalities of (the result of) the executed operators while the content module does not. The Refine module merges the embedding produced by cardinality and content, and processes the remaining operators (which are not executed) for the final result. We compare LPCE-R with two alternative (perhaps more natural) designs: (1) LPCE-R-*Single* uses only one module sharing the same structure as the cardinality module in LPCE-R, which has access to the cardinalities of intermediate results. The real cardinalities are used for training while for 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*. Among the three models, LPCE-R-*Single* has the worst performance in most cases. This is because LPCE-R-*Single* is trained using the real cardinalities but uses the estimated cardinalities of the remaining operators for inference. As the estimations are inaccurate, errors will accumulate along the join tree in the model. LPCE-R-*Two* performs better because its refine module does not rely on cardinality estimations for the remaining operators. 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

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

Executed operators	LPCE-R						LPCE-R-*Single*						LPCE-R-*Two*					
	50th	75th	95th	99th	max	mean	50th	75th	95th	99th	max	mean	50th	75th	95th	99th	max	mean
4	**7.15**	**21.6**	**66.3**	**1203**	**8161**	**72.6**	17.1	83.1	396	5377	81022	433	10.0	37.0	123	3006	11778	123
8	**6.93**	**18.9**	**65.8**	**1877**	**5743**	**67.8**	12.6	68.9	314	5294	20536	247	9.11	33.9	102	2956	11223	88.3
12	**5.41**	**16.2**	**56.9**	**1561**	**3820**	**51.7**	8.69	45.1	148	2688	13139	156	7.16	25.3	97.8	1921	8133	64.8



(a) Time decomposition for training LPCE (b) Time decomposition for model training

Figure 25: Time decomposition of LPCE training cost (data fake)

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. Therefore, in LPCE-R, module cardinality complements module content by providing more information about the executed operators.

Training cost: Making LPCE requires the efforts of sample collection and model training. We report the time decomposition of training cost in Figure 25. The time of sample collection nearly linearly increase with the number of collected samples. When LPCE-I is trained using 10,000 sample queries, and error stabilizes after about 50 epochs for the test queries. Module content and cardinality of LPCE-R are pre-trained using the same procedure as LPCE-I. Module refine is fine-tuned based on module content and cardinality, and can converge within 10 epochs. The model training time is about 0.83 hours, and the time of collecting 10,000 sample is 10.4 hours.

9 CONCLUSION

We propose LPCE, a learning-based cardinality estimator that progressively refines the cardinality estimations during the query execution process. We observed that to reduce the end-to-end execution time of queries, both fast inference and high accuracy are necessary but no existing cardinality estimators can satisfy the two requirements simultaneously. 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 refinement model with three modules to instantiate LPCE. Extensive experiments show that our LPCE outperforms existing learning-based estimators in reducing the end-to-end execution time of queries, especially in worst cases. There are many interesting problems in applying learning-based progressive cardinality estimation to speed up query execution, including devising tailored plan enumeration algorithms and improving the performance of data-driven or hybrid learning-based estimators, which we leave for future work.

10 APPENDIX

10.1 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 26, 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, 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.

10.2 Example of query optimization with LPCE

Figure 27 show an query example with our LPCE. Without the cardinality estimation provided by LPCE, PostgreSQL severely underestimates the cardinality of each join, and the last join thus is evaluated by nested loop but turns out the long execution time. LPCE-I provides accurate estimation, and thus avoids the wrong choice of nest loop, decreasing the execution time significantly by 97.0%. However, the estimation is still large, over the threshold 50 of triggering re-optimization at the second join (i.e., estimated and real cardinality are 991,975 and 3,528 respectively). The checkpoint

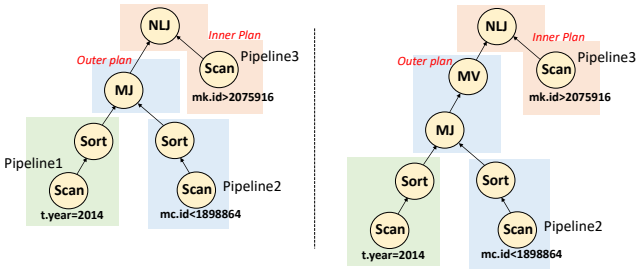


Figure 26: The nested loop join in pipeline processing of PostgreSQL (left), and modified blocked outer side to support re-optimization (right).

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
```

at the join detects the error, and then invokes LPCE-R to refine the cardinalities and adjust the plan. The query reuses the done work of the checkpoint, and continues executing with a modified plan, further speedup the execution time by 47.0%.

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
```

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] Yu Chen and Ke Yi. 2017. Two-Level Sampling for Join Size Estimation. In SIGMOD. 759–774.
[6] Amol Deshpande, Minos N. Garofalakis, and Rajeev Rastogi. 2001. Independence is Good: Dependency-Based Histogram Synopses for High-Dimensional Data. In SIGMOD. 199–210.
[7] Anshuman Dutt and Jayant R Haritsa. 2014. Plan bouquets: query processing without selectivity estimation. In SIGMOD. 1039–1050.
[8] 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.
[9] 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.
[10] Sumit Ganguly, Phillip B. Gibbons, Yossi Matias, and Abraham Silberschatz. 1996. Bifocal Sampling for Skew-Resistant Join Size Estimation. In SIGMOD. 271–281.
[11] Mathieu Germain, Karol Gregor, Iain Murray, and Hugo Larochelle. 2015. MADE: Masked Autoencoder for Distribution Estimation. In ICML. 881–889.
[12] Dimitrios Gunopulos, George Kollios, Vassilis J. Tsotras, and Carlotta Domeniconi. 2005. Selectivity estimators for multidimensional range queries over real attributes. VLDB J. 14, 2 (2005), 137–154.
[13] 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.
[14] 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.
[15] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. arXiv preprint arXiv:1503.02531 (2015).
[16] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. Neural computation 9, 8 (1997), 1735–1780.

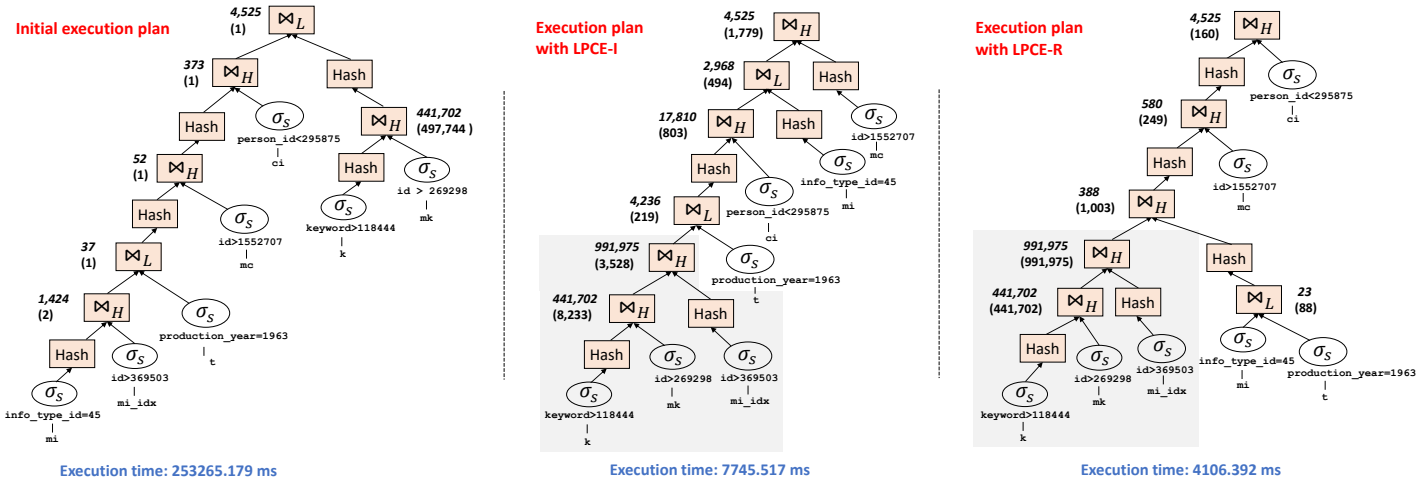


Figure 27: Examples of query re-optimization with LPCE, where the upright and italic numbers indicate estimated and real cardinalities, respectively. The gray marked area denotes the executed operators while the star marks the selected execution plan

[17] Navin Kabra and David J DeWitt. 1998. Efficient mid-query re-optimization of sub-optimal query execution plans. In SIGMOD. 106–117.
[18] 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.
[19] 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.
[20] Tao Lei, Yu Zhang, and Yoav Artzi. 2017. Training RNNs as Fast as CNNs. arXiv preprint arXiv:1709.02755 (2017).
[21] 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.
[22] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. 2017. Cardinality Estimation Done Right: Index-Based Join Sampling. In CIDR.
[23] 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.
[24] 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.
[25] Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, and Miso Cilimdzic. 2004. Robust query processing through progressive optimization. In SIGMOD. 659–670.
[26] 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).
[27] Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. 2015. FitNets: Hints for Thin Deep Nets. In ICLR.
[28] Michael Stillger, Guy M Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO-DB2's learning optimizer. In VLDB, Vol. 1. 19–28.
[29] Ji Sun and Guoliang Li. 2019. An End-to-End Learning-based Cost Estimator. PVLDB 13, 3 (2019), 307–319.
[30] TPCH. 2022. TCP-H Benchmark. (March 2022). https://www.tpc.org/tpch/
[31] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. arXiv preprint arXiv:1706.03762 (2017).
[32] David Vengerov, Andre Cavalheiro Menck, Mohamed Zaït, and Sunil Chakkappen. 2015. Join Size Estimation Subject to Filter Conditions. PVLDB 8, 12 (2015), 1530–1541.
[33] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2020. Are We Ready For Learned Cardinality Estimation? arXiv preprint arXiv:2012.06743 (2020).
[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] Wentao Wu, Jeffrey F Naughton, and Harneet Singh. 2016. Sampling-based query re-optimization. In SIGMOD. 1721–1736.
[36] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: one cardinality estimator for all tables. arXiv preprint arXiv:2006.08109 (2020).
[37] 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.
[38] Feng Yu, Wen-Chi Hou, Cheng Luo, Dunren Che, and Mengxia Zhu. 2013. CS2: a new database synopsis for query estimation. In SIGMOD. 469–480.
[39] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Ruslan Salakhutdinov, and Alexander Smola. 2017. Deep sets. arXiv preprint arXiv:1703.06114 (2017).