Improved Cardinality Estimation by Learning Queries Containment Rates

Rojeh Hayek
CS Department, Technion
Haifa 3200003, Israel
srojeh@cs.technion.ac.il

Oded Shmueli
CS Department, Technion
Haifa 3200003, Israel
oshmu@cs.technion.ac.il

ABSTRACT

The *containment rate* of query $Q1$ in query $Q2$ over database D is the percentage of $Q1$'s result tuples over D that are also in $Q2$'s result over D. We directly estimate containment rates between pairs of queries over a specific database. For this, we use a specialized deep learning scheme, Containment Rate Network (CRN), which is tailored to representing pairs of SQL queries (inspired by the MSCN model [22]). Result-cardinality estimation is a core component of query optimization. We describe a novel approach for estimating queries' result-cardinalities using estimated containment rates among queries. This containment rate estimation may rely on CRN or embed, unchanged, known *cardinality* estimation methods. Experimentally, our novel approach for estimating cardinalities, using containment rates between queries, on a challenging real-world database, realizes significant improvements to state of the art cardinality estimation methods.

1 INTRODUCTION

Query $Q1$ is contained in (resp. equivalent to), query $Q2$, analytically, if for all database states D, $Q1$'s result over D is contained in (resp., equals) $Q2$'s result over D. Query containment is a well-known concept that has applications in query optimization. It has been extensively researched in database theory, and many algorithms were proposed for determining containment under different assumptions [8, 9, 16, 40]. However, determining query containment analytically is not practically sufficient. Two queries may be analytically unrelated by containment, although, the execution result on a *specific* database of one query may actually be contained in the other. For example, consider the queries:
Q1: *select * from movies where title = 'Titanic'*
Q2: *select * from movies where release = 1997 and director = 'James Cameron'*
Both queries execution results are identical since there is only one movie called Titanic that was released in 1997 and directed by James Cameron (he has not directed any other movie in 1997). Yet, using the analytic criterion, the queries are unrelated at all by containment.

To our knowledge, while query containment and equivalence have been well researched in past decades, determining the containment rate between two queries on a *specific* database, has not been considered by past research.

By definition, the containment rate of query $Q1$ in query $Q2$ on database D is the percentage of rows (tuples) in $Q1$'s execution result over D that are also in $Q2$'s execution result over D. Determining containment rates allows us to solve other problems, such as determining equivalence between two queries, or whether one query is fully contained in another, on the same *specific* database. In addition, containment rates can be used in many practical applications, for instance, query clustering, query recommendation [11, 15], and in cardinality estimation as will be described subsequently.

Our approach for estimating containment rates is based on a specialized deep learning model, CRN, which enables us to express query features using sets and vectors. An input query is converted into three sets, T, J and P representing the query's tables, joins and column predicates, respectively. Each element of these sets is represented by a vector. Using these vectors, CRN generates a single vector that represents the whole input query. Finally, CRN estimates the containment rate of two input queries by using their representative vectors as input to another specialized neural network. Thus, the CRN model relies on the ability of the neural network to learn the vector representation of queries relative to the *specific* database. As a result, we obtain a small and accurate model for estimating containment rates.

In addition to the CRN model, we introduce a novel technique for estimating queries' cardinalities using estimated query containment rates. We show that using the proposed technique we improve current cardinality estimation techniques significantly. This is especially the case when there are multiple joins, where the known cardinality estimation techniques suffer from under-estimated results and errors that grow exponentially as the number of joins increases [14]. Our technique estimates the cardinalities more robustly (x150/x175 with 4 joins queries, and x1650/x120 with 5 joins queries, compared with PostgreSQL and MSCN, respectively).

As shown in [26], to obtain an efficient query plan, the query optimizer chooses the cheapest alternative from semantically equivalent plan alternatives. Since the cost model uses the cardinality estimates as a principal input, the more accurate the cardinality estimates are, the more accurate the predicted plans costs are. Thus, by using the more accurate cardinality estimates obtained from our technique, the query optimizer can generate better query plans, resulting in faster query execution time.

We compare our technique with PostgreSQL [1], and the pioneering multi-set convolutional network (MSCN) model [22], by examining, on the real-world IMDb database [26], join crossing correlations queries which are known to present a tough challenge to cardinality estimation methods [26, 28, 35].

We show that by employing known existing cardinality estimation methods for containment estimation, we can improve on their cardinality estimates as well, without changing the methods themselves. Thus, our novel approach is highly promising for solving the cardinality estimation problem, the "Achilles heel" of query optimization [30], a cause of many performance issues [26].

The rest of this paper is organized as follows. In Section 2 we define the containment rate problem and in Sections 3-4 we describe and evaluate the CRN model for solving this problem. In Sections 5-6 we describe and evaluate our new approach for

estimating cardinalities using containment rates. In Section 7 we show how one can adapt the new ideas to improve existing cardinality estimation models, and in Section 8 we compare the prediction time among the different approaches. Finally, Sections 9-10 present related work, conclusions and future work.

2 CONTAINMENT RATE DEFINITION

We define the containment rate between two queries $Q1$, and $Q2$ on a *specific* database D. *Query $Q1$ is x%-contained in query $Q2$ on database D if precisely x% of $Q1$'s execution result rows on database D are also in $Q2$'s execution result on database D.* The containment rate is formally a function from **QxQxD** to **R**, where **Q, D** and **R** are the set of all queries, all databases, and the Real numbers, respectively. This function can be directly calculated as follows. Let $Q1(D) = (A, m_1)$ and $Q2(D) = (B, m_2)$ be multisets[1] representing queries $Q1$ and $Q2$ execution results on database D, respectively, then:

$$x\% = \frac{\sum_{x \in (A \cap B)} m_1(x)}{\sum_{x \in A} m_1(x)} * 100$$

Where operator $\cap$ is the regular set intersection operator (in case $Q1$'s execution result is empty, then $Q1$ is 0%-contained in $Q2$). Note that the containment rate is defined only on pairs of queries whose SELECT and FROM clauses are *identical*.

Since we aim to estimate cardinalities using containment rates, we consider only queries with SELECT * clauses, then, given a query Q whose SELECT clause includes specific columns, Q's cardinality is identical to the cardinality of the query with a SELECT * clause instead (as long as the *DISTINCT* keyword is not used). Therefore, in practice, the requirement that the clauses need to be *identical* applies only to the FROM clauses.

2.1 Containment Rate Operator

We denote the containment rate *operator* between queries $Q1$ and $Q2$ on database D as:

$$Q1 \subset_{\%}^{D} Q2$$

Operator $\subset_{\%}^{D}$ returns the containment rate between the given input queries on database D. That is, $Q1 \subset_{\%}^{D} Q2$ returns $x\%$, if $Q1$ is $x\%$-contained in query $Q2$ on database D. For simplicity, we do not mention the *specific* database, as it is usually clear from context. Hence, we write the containment rate operator as $\subset_{\%}$.

3 LEARNED CONTAINMENT RATES

From a high-level perspective, applying machine learning to the containment rate estimation problem is straightforward. Following the training of the CRN model with pairs of queries $(Q1, Q2)$ and the actual containment rates $Q1 \subset_{\%} Q2$, the model is used as an estimator for other, unseen pairs of queries. (Later on, as described in Section 5, we will make use of this model to estimate cardinalities of single queries). There are, however, several questions whose answers determine whether the machine learning model (CRN) will be successful. (1) Which supervised learning algorithm/model should be used. (2) How to represent queries as input and the containment rates as output to the model ("featurization"). (3) How to obtain the initial training dataset ("cold start problem"). Next, we describe how we address each one of these questions.

[1]From Wikipedia: A multiset may be formally defined as a 2-tuple (S, m) where S is the underlying set of the multiset, formed from its distinct elements, and $m : S \rightarrow N_{\geq 1}$ is a function from S to the set of the positive integers, giving the multiplicity. The number of occurrences of element x in the multiset is $m(x)$.

3.1 Cold Start Problem

3.1.1 Defining the Database. We generated a training-set, and later on evaluated our model on it, using the IMDb database. IMDb contains many correlations and has been shown to be very challenging for cardinality estimators [26]. This database contains a plethora of information about movies and related facts about actors, directors, and production companies, with more than 2.5M movie titles produced over 130 years (starting from 1880) by 235,000 different companies with over 4M actors.

3.1.2 Generating the Development Dataset. Our approach for solving the "cold start problem" is to obtain an initial training corpus using a specialized queries generator that randomly generates queries based on the IMDB schema and the actual columns values. Our queries generator generates the dataset in three main steps. In the first step (similarly to MSCN's queries generator), it repeatedly generates multiple SQL queries as follows. It randomly chooses a set of tables t ($t = \{bt_1, bt_2, ..., bt_{|t|}\}$). Then, it adds $|t| - 1$ join edges to the query, $bt_i.col_a = bt_{i+1}.col_b$, $1 \leq i < |t|$. Each of these joins is on a column containing the ID of movies (each table in IMDB has such a column). Note that when $|t| = 1$, there are no joins in the query.

For each base table bt in t, it uniformly draws the number of query predicates p_{bt} ($0 \leq p_{bt} \leq$ number of columns in table bt). Subsequently, for each predicate it uniformly draws a column from the relevant table bt, a predicate type ($<$, $=$, *or* $>$), and a value from the corresponding column values range in the database. To avoid a combinatorial explosion, and to simplify the problem that the model needs to learn, we force the queries generator to create queries with up to two joins and let the model generalize to a larger number of joins (that is, the maximum cardinality of set t is 3). Note that all the generated queries include a SELECT * clause. They are denoted as *initial-queries*.

To create pairs of queries that are contained in each other with different containment rates, we generate, in the second step, queries that are "similar" to the *initial-queries*, but still, different from them, as follows. For each query Q in *initial-queries*, the generator repeatedly creates multiple queries by randomly changing query Q's predicates' types, or the predicates' values, and by randomly adding additional predicates to the original query Q. This way, we create a "hard" dataset, which includes pairs of queries that look "similar", but having mutual containment rates that vary significantly. Finally, in the third and last step, using the queries obtained from both previous steps, the queries generator generates pairs of queries whose FROM clauses are identical.

After generating the dataset, we execute the dataset queries on the IMDb database, to obtain their true containment rates and skip query pairs that include a query with an empty result set. Using this process, we obtain an initial training set of 100,000 pairs of queries with zero to two joins. We split the training samples into 80% training samples and 20% validation samples.

3.2 Model

Featurizing all the queries' literals and predicates as one "big hot vector", over all the possible words that may appear in the queries, is impractical. Also, serializing the queries' SELECT, FROM, and WHERE clauses elements into an ordered sequence of elements, is not practical, since the order in these clauses is arbitrary. Thus, standard deep neural network architectures such as simple multi-layer perceptrons [6], convolutional neural networks [6], or recurrent neural networks [6], are not directly applicable to our problem.

Our *Containment Rate Network* (CRN) model uses a specialized vector representation for representing the input queries and the output containment rates. As depicted in Figure 1, the CRN model runs in three main stages. Consider an input queries pair $(Q1, Q2)$. In the first stage, we convert $Q1$ (resp., $Q2$) into a set of vectors $V1$ (resp., $V2$). Thus $(Q1, Q2)$ is represented by $(V1, V2)$. In the second stage, we convert set $V1$ (resp., $V2$) into a unique single representative vector $Qvec1$ (resp., $Qvec2$), using a specialized neural network, MLP_i, for each set separately. In the third stage, we estimate the containment rate $Q1 \subset_{\%} Q2$, using the representative vectors $Qvec1$ and $Qvec2$, and another specialized neural network, MLP_{out}.

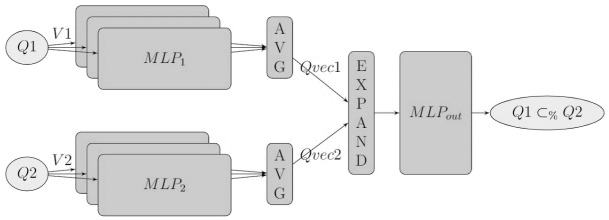

Figure 1: CRN Model Archeticture.

3.2.1 First Stage, from $(Q1, Q2)$ *to* $(V1, V2)$. In the same way as MSCN model [22], we represent each query Q as a collection of three sets (T, J, P). T is the set of all the tables in Q's FROM clause. J is the set of all the joins (i.e., join clauses) in Q's WHERE clause. P is the set of all the (column) predicates in Q's WHERE clause. Using sets T, J, and P, we obtain a set of vectors V representing the query, as described later. Unlike MSCN, in our model all the vectors of set V have the same dimension and the same segmentation as depicted in Table 1, where $\#T$ is the number of all the tables in the database, $\#C$ is the number of all the columns in all the database tables, and $\#O$ is the number of possible predicates operators. In total, the vector dimension is $\#T + 3 * \#C + \#O + 1$, denoted as L.

The queries tables, joins and column predicates (sets T, J and P) are inseparable, hence, treating each set individually using different neural networks may disorientate the model. Therefore, we choose to featurize these sets using the same vector format in order to ease learning.

Type	Table	Join		Column Predicate		
Segment	T-seg	J1-seg	J2-seg	C-seg	O-seg	V-seg
Segment size	$\#T$	$\#C$	$\#C$	$\#C$	$\#O$	1
Featurization	*one hot*	*one hot*	*one hot*	*one hot*	*one hot*	*norm*

Table 1: Vector Segmentation.

Element of sets T, J, and P, are represented by vectors as follows (see a simple example in Figure 2). All the vectors have the same dimension L. Each table $t \in T$ is represented by a unique one-hot vector (a binary vector of length $\#T$ with a single non-zero entry, uniquely identifying a specific table) placed in the T-seg segment. Each join clause of the form $(col1, =, col2) \in J$ is represented as follows. $col1$ and $col2$ are represented by a unique one-hot vectors placed in J1-seg and J2-seg segments, respectively. Each predicate of the form $(col, op, val) \in P$ is represented as follows. col and op are represented by a unique one-hot vectors placed in the C-seg and V-seg segments, respectively. val is represented as a normalized value $\in [0, 1]$, normalized using the minimum and maximum values of the respective column, placed in the V-seg segment. For each vector, all the other unmentioned segments are zeroed. Given input queries pair, $(Q1, Q2)$, we convert query $Q1$ (resp., $Q2$) into sets T, J and P, and *each* element of these sets is represented by a vector as described above, together generating set $V1$ (resp., $V2$).

3.2.2 Second Stage, from $(V1, V2)$ *to* $(Qvec1, Qvec2)$. Given set of vectors V_i, we present each vector of the set into a fully-connected one-layer neural network, denoted as MLP_i, converting each vector into a vector of dimension H. The final representation $Qvec_i$ for this set is then given by the average over the individual transformed representations of its elements, i.e.,

$$Qvec_i = \frac{1}{|V_i|} \sum_{v \in V_i} MLP_i(v)$$

$$MLP_i(v) = Relu(vU_i + b_i)$$

Where $U_i \in R^{LxH}$, $b_i \in R^H$ are the learned weights and bias, and $v \in R^L$ is the input row vector. We choose an average (instead of, e.g., sum) to ease generalization to different numbers of elements in the sets, as otherwise the overall magnitude of $Qvec$ would vary depending on the number of elements in the set V_i.

3.2.3 Third Stage, from $(Qvec1, Qvec2)$ *to* $Q1 \subset_{\%} Q2$. Given the representative vectors of the input queries,$(Qvec1, Qvec2)$, we aim to predict the containment rate $Q1 \subset_{\%} Q2$ as accurately as possible. Since we do not know what a "natural" containment rate measure is in the representative queries vector space, encoded by the neural networks of the second step, we use a fully-connected two-layer neural network, denoted as MLP_{out}, to compute the estimated containment rate of the input queries, leaving it up to this neural network to learn the correct containment rate measure.

MLP_{out} takes as input a vector of size $4H$ which is constructed using function *ExpandFunction* that creates a row of concatenated vectors of size $4H$ using vectors $Qvec1$ and $Qvec2$. We use this function in order to provide the final network, MLP_{out}, with additional information that may enhance its learning and thus obtain more accurate containment rates estimations.

The first layer in MLP_{out} converts the input vector into a vector of size $2H$. The second layer converts the obtained vector of size $2H$, into a single value representing the containment rate.

$$\hat{y} = MLP_{out}(Expand(Qvec1, Qvec2))$$

$$MLP_{out}(v) = Sigmoid(ReLU(vU_{out1} + b_{out1})U_{out2} + b_{out2})$$

$$Expand(v_1, v_2) = [v_1,\ \ v_2,\ \ abs(v_1 - v_2),\ \ v_1 \odot v_2]$$

Here, $\hat{y}$ is the estimated containment rate (a number in $[0, 1]$), $U_{out1} \in R^{4Hx2H}$, $b_{out1} \in R^{2H}$ and $U_{out2} \in R^{2Hx1}$, $b_{out2} \in R^1$ are the learned weights and bias, abs is the absolute value function, and $\odot$ is the dot-product function.

We use the $ReLU$[2] activation function for hidden layers in all the neural networks, as they show strong empirical performance advantages and are fast to evaluate.

In the final step, we apply the $Sigmoid$[3] activation function in the second layer to output a float value in the range [0,1], as the containment rate values are within this interval. Therefore, we do not apply any featurization on the containment rates (the output of the model) and the model is trained with the actual containment rate values without any featurization steps.

[2] ReLU(x) = max(0,x); see [36].
[3] Sigmoid(x) = $1/(1 + e^{-x})$; see [36].

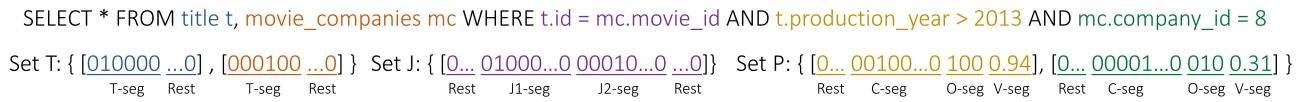

Figure 2: Query featurization as sets of feature vectors obtained from sets T, J and P (Rest denotes zeroed segments).

3.2.4 Loss Function. Since we are interested in minimizing the ratio between the predicted and the actual containment rates, we use the q-error metric in our evaluation. We train our model to minimize the mean q-error [33], which is the ratio between an estimated and the actual containment rate (or vice versa). Let y be the true containment rate, and $\hat{y}$ the estimated rate, then the q-error is defined as follows.

$$q-error(y,\hat{y}) \;=\; \hat{y} > y \;?\; \frac{\hat{y}}{y} \;:\; \frac{y}{\hat{y}}$$

The q-error is not defined when y (or y') equals zero. Therefore, in creating the training and testing datasets we skip query pairs that include a query with an empty results set (see Section 3.1.2).

In addition to optimizing the mean q-error, we also examined the mean squared error (MSE) and the mean absolute error (MAE) as optimization goals. MSE and MAE would optimize the squared/absolute differences between the predicted and the actual containment rates. Optimizing with theses metrics makes the model put less emphasis on heavy outliers (that lead to large errors). Therefore, we decided to optimize our model using the q-error metric which yielded better results.

3.3 Training and Testing Interface

Building CRN involves two main steps. (1) Generating a random training set using the schema and data information as described in Section 3.1. (2) Repeatedly using this training data, we train the CRN model as described in Section 3.2 until the mean q-error of the validation test starts to converges to its best absolute value. That is, we use the early stopping technique [39] and stop the training before convergence to avoid over-fitting. Both steps are performed on an immutable snapshot of the database.

After the training phase, to predict the containment rate of an input query pair, the queries first need to be transformed into their feature representation, and then they are presented as input to the model, and the model outputs the estimated containment rate (Section 3.2). We train and test our model using the Tensor-Flow framework [34], and make use of the efficient Adam optimizer [21] for training the model.

3.4 Hyperparameter Search

To optimize our model's performance, we conducted a search over its hyperparameter space. In particular, we focused on tuning the neural networks hidden layer size (H) as we found out that this hyperparameter has the most impact on the results.

Note that the same H value is shared in all the neural networks of the CRN model, as described in section 3.2. During the tuning of the size hyperparameter of the neural network hidden layer, we found that increasing the size of our hidden layer generally led to an increase in the model accuracy, till it reached the best mean q-error on the validation test. Afterwards, the results began to decline in quality because of over-fitting (see Figure 3). Hence, we choose a hidden layer of size 512, as a good balance between accuracy and training time.

Overall, we found that our model performs uniformly well across a wide range of settings when considering different batch sizes and learning rates. We use a learning rate of 0.001, and batch size of 128, as these settings lead to the best results on the validation test.

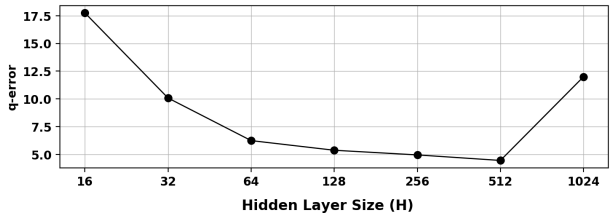

Figure 3: The mean q-error on the validation set with different hidden layer sizes.

3.5 Model Computational Costs

We analyze the training, prediction, and space costs of the CRN model with the default hyperparameters (H=512, batch size=128, learning rate=0.001).

3.5.1 Training Time. Figure 4 shows how the mean q-error of the validation set decreases with additional epochs, until convergence to a mean q-error of around 4.5. The CRN model requires almost 120 passes on the training set to converge. On average, measured across six runs, a training run with 120 epochs takes almost 200 minutes.

3.5.2 Prediction Time. The prediction process is dominated by converting the input queries into the corresponding vectors, and presenting these vectors as input to the CRN model. On average, the prediction time is 0.5ms per single pair of queries, including the overhead introduced by the Tensor-Flow framework.

3.5.3 Model Size. The CRN model includes all the learned parameters mentioned in Section 3.2 (U_1, U_2, U_{out1}, U_{out2}, b_1, b_2, b_{out1}, b_{out2}). In total, there are $2*L*H+8*H^2+6*H+1$ learned parameters. In practice, the size of the model, when serialized to disk, is roughly $1.5MB$.

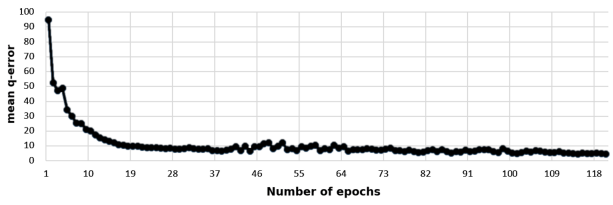

Figure 4: Convergence of the mean q-error on the validation set.

4 CONTAINMENT EVALUATION

Since the focus of this paper is on cardinality estimation using containment rates, in this section we only briefly present the containment evaluation results of the CRN model when compared to other (baseline) methods. In the following sections, we present in detail the experiments of the cardinality estimation technique.

Since to the best of our knowledge, the problem of determining containment rate has not been addressed till now, we use a transformation as described in Section 4.1 below.

4.1 From Cardinality to Containment

To our knowledge, this is the first work to address the problem of containment rate estimation. In order to compare our results with different baseline methods, we used existing cardinality estimation methods to predict the containment rates, using the Crd2Cnt transformation, as depicted in the middle part diagram in Figure 5. (This transformation will be used also in our technique to improve existing cardinality estimation models in Section 7).

4.1.1 The Crd2Cnt Transformation. Given a cardinality estimation model M, we can convert it to a containment rate estimation model using the Crd2Cnt transformation which returns a model M' for estimating containment rates. The obtained model M' functions as follows. Given input queries $Q1$ and $Q2$, whose containment rate $Q1 \subset_{\%} Q2$ needs to be estimated:

- Calculate the cardinality of query $Q1 \cap Q2$ using M.
- Calculate the cardinality of query $Q1$ using M.
- Then, the containment rate estimate is:

$$Q1 \subset_{\%} Q2 = \frac{|Q1 \cap Q2|}{|Q1|}$$

Here, $Q1 \cap Q2$ is the intersection query of $Q1$ and $Q2$ whose SELECT and FROM clauses are identical to $Q1$'s (or $Q2$'s) clauses, and whose WHERE clause is $Q1$'s AND $Q2$'s WHERE clauses. Note that, by definition, if $|Q1| = 0$ then $Q1 \subset_{\%} Q2 = 0$.

Given model M, we denote the obtained model M', via the Crd2Cnt transformation, as Crd2Cnt(M).

4.2 Experimental Results

We compared the CRN model predictions to those based on the other examined cardinality estimation models, using the Crd2Cnt transformation. We evaluated the models with several workloads, that included over 2000 queries with zero to five joins, on the challenging real-world IMDB database [26]. In terms of mean q-error [33], the CRN model reduced the mean q-errors by a factor of roughly 8 compared with the estimates obtained from Crd2Cnt(PostgreSQL) and Crd2Cnt(MSCN).

To provide a fuller picture, in Table 2 we show the percentiles, maximum, and mean q-errors, on one of the examined evaluation workloads. Additional details may be found in *arXiv* [18].

	50th	75th	90th	95th	99th	max	mean
Crd2Cnt(PostgreSQL)	4.5	46.22	322	1330	39051	316122	1345
Crd2Cnt(MSCN)	4.1	17.85	157	754	14197	768051	1238
CRN	**3.64**	**13.19**	**96.6**	**255**	**2779**	**56965**	**161**

Table 2: Estimation errors on 1200 examined queries with zero to five joins, equally distributed in the number of joins. In all the similar tables presented in this paper, we provide the percentiles, maximum, and the mean q-errors of the tests. The *p*'th percentile, is the q-error value below which *p*% of the test q-errors are found. For example, 50% of the CRN test q-errors are smaller than 3.64.

5 CARDINALITY ESTIMATION USING CONTAINMENT RATES

In this section we consider one application of the proposed containment rate estimation model: cardinality estimation. We introduce a novel approach for estimating cardinalities using query containment rates, and we show that using the proposed approach, we improve cardinality estimations significantly, especially in the case when there are multiple joins.

A traditional query optimizer is crucially dependent on cardinality estimation, which enables choosing among different plan alternatives by using the cardinality estimation of intermediate results within query execution plans. Therefore, the query optimizer must use reasonably good estimates. However, estimates produced by all widely-used database cardinality estimation models are routinely significantly wrong (under/over-estimated), resulting in not choosing the best plans, leading to slow executions [26].

Three principal approaches for estimating cardinalities have emerged. (1) Using database profiling [1]. (2) Using histograms [3, 7]. (3) Using sampling techniques [5, 27, 37]. Recently, deep learning neural networks were also used for solving this problem [22, 45]. However, all these approaches, with all the many attempts to improve them, have conceptually addressed the problem *directly* in the same way, as a black box, where the input is a query, and the output is its cardinality estimation, as described in the leftmost diagram in Figure 5. In our proposed approach, we address the problem differently, and we obtain better estimates as described in Section 6.

In prior works, the answers to previous queries were used for speeding up new queries, by incrementally updating histograms, and in the context of query re-optimization [3, 7, 13, 20]. Similarly, using the CRN model for predicting containment rates, we are making use of these previous answers to reveal the underlying relations between the new queries and the previous ones.

Our new technique for estimating cardinalities mainly relies on two key ideas. The first one is the new framework in which we solve the problem. The second is the use of a *queries pool* that maintains multiple previously executed queries along with their actual *cardinalities*, as part of the database meta information. The queries pool provides new information that enables our technique to achieve better estimates. Using a containment rate estimation model, we make use of previously executed queries along with their actual cardinalities to estimate the result-cardinality of a new query. This is done with the help of a simple transformation from the problem of containment rate estimation to the problem of cardinality estimation (see Section 5.1).

5.1 From Containment to Cardinality

Using a containment rate estimation models, we can obtain cardinality estimates using the Cnt2Crd transformation, as depicted in the rightmost diagram in Figure 5.

5.1.1 The Cnt2Crd Transformation. Given a containment rate estimation model[4] M, we convert it to a cardinality estimation model using the Cnt2Crd transformation which returns a model M' for estimating cardinalities. The obtained model M' functions as follows. We are given a "new" query, denoted as Q_{new}, as input to cardinality estimation. Assume that there is an "old" query, denoted as Q_{old}, whose FROM clause is the same as Q_{new}'s

[4]The term "model" may refer to an ML model or simply to a method.

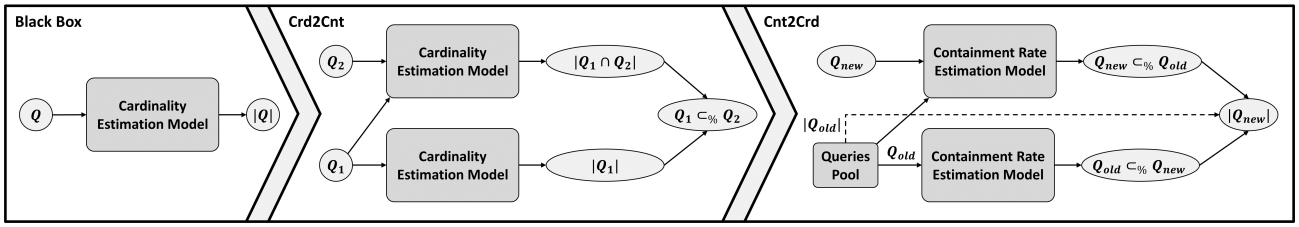

Figure 5: A novel approach, from cardinality estimation to containment rate estimation, and back to cardinality estimation by using a queries pool.

FROM clause, that has already been executed over the database, and therefore $|Q_{old}|$ is known, then M' functions as follows:

- Calculate $x\_rate = Q_{old} \subset_{\%} Q_{new}$ using M.
- Calculate $y\_rate = Q_{new} \subset_{\%} Q_{old}$ using M.
- Then, the cardinality estimate equals to:

$$|Q_{new}| = \frac{x\_rate}{y\_rate} * |Q_{old}|$$

provided that $y\_rate = Q_{new} \subset_{\%} Q_{old} \neq 0$. This is true, since:

$$x\_rate = \frac{|Q_{new} \cap Q_{old}|}{|Q_{old}|}, \quad y\_rate = \frac{|Q_{new} \cap Q_{old}|}{|Q_{new}|}$$

And therefore,

$$\frac{x\_rate}{y\_rate} = \frac{|Q_{new} \cap Q_{old}|}{|Q_{old}|} * \frac{|Q_{new}|}{|Q_{new} \cap Q_{old}|} = \frac{|Q_{new}|}{|Q_{old}|}$$

where the query intersection operator, $\cap$, is as defined in Section 4.1.1. Given model M, we denote the obtained model M', via the Cnt2Crd transformation, as Cnt2Crd(M).

5.2 Queries Pool

Our technique for estimating cardinality relies mainly on a queries pool that includes records of multiple queries.

The queries pool is envisioned to be an additional component of the DBMS, along with all the other customary components. It includes multiple queries with their actual cardinalities[5], but without the queries execution results. Therefore, holding such a pool in the DBMS as part of its meta information does not require significant storage space or other computing resources. Maintaining a queries pool in the DBMS is thus a reasonable expectation. The DBMS continuously executes queries, and therefore, we can easily configure the DBMS to store these queries along with their actual cardinalities in the queries pool.

In addition, we may construct in advance a queries pool using a queries generator that randomly creates multiple queries with many of the possible joins, and with different column predicates. We then execute these queries on the database to obtain and save their actual cardinalities in the queries pool.

Notice that we can combine both approaches (actual computing and a generator) to create the queries pool. The advantage of the first approach is that in a real-world situation, queries that are posed in sequence by the same user, may be similar and therefore we can get more accurate cardinality estimates. The second approach helps in cases where the queries posed by users are diverse (e.g., different FROM clauses). Therefore, in such cases, we need to make sure, in advance, that the queries pool contains sufficiently many queries that cover all the possible cases.

[5]Due to limited space, we do not detail the efficient hash-based data structures used to implement the queries pool.

Given a query Q whose cardinality is to be estimated , it is possible that we fail to find any appropriate query, in the queries pool, to match with query Q. This happens when all the queries in the queries pool have a different FROM clause than that of query Q, or that they are not contained at all in query Q. In such cases we can always use the *known* basic cardinality estimation models. In addition, we can make sure that the queries pool includes queries with the most frequently used FROM clauses, with empty column predicates. That is, queries of the following form:

$$SELECT\ *\ FROM\ -set\ of\ tables-\ WHERE\ TRUE$$

In this case, for most of the queries posed in the database, there is at least one query that matches in the queries pool with the given query, and hence, we can estimate the cardinality (perhaps less accurately) without resorting to the basic cardinality estimation models.

5.3 A Cardinality Estimation Technique

Consider a new query Q_{new}, and assume that the DBMS includes a queries pool as previously described. To estimate the cardinality of Q_{new} accurately, we use *multiple* "old" queries instead of *one* query, using the same Cnt2Crd transformation of Section 5.1.1, as described in Figure 6.

```
EstimateCardinality(Query Q_new, Queries Pool QP):
    results = empty list

    For every pair (Q_old, |Q_old|) in QP:
        if Q_old's FROM clause ≠ Q_new's FROM clause:
            continue
        Calculate x_rate = Q_old ⊂% Q_new
        Calculate y_rate = Q_new ⊂% Q_old
        if y_rate <= epsilon: /* y essentially zero */
            continue
        results.append(x_rate/y_rate * |Q_old|)

    return F(results)
```

Figure 6: Cardinality Estimation Technique.

Algorithm EstimateCardinality considers all the *matching* queries whose FROM clauses are identical to Q_{new}'s FROM clause. For each matching query, we estimate Q_{new}'s cardinality using the Cnt2Crd transformation and save the estimated result in the *results* list. The final cardinality is obtained by applying the final function, F, that converts all the estimated results recorded in the *results* list, into a single final estimation value. Note that the technique can be easily parallelized since each iteration in the For loop is independent, and thus can be calculated in parallel.

5.3.1 Comparing Different Final Functions. We examined various final functions (F), including:

- Median, returning the median value of the *results* list.
- Mean, returning the mean value of the *results* list.
- Trimmed mean, returning the trimmed mean of the *results* list without the 25% outliers (trimmed removes a designated percentage of the largest and smallest values before calculating the mean).

Experimentally, the cardinality estimates using the various functions were very similar in terms of q-error. But the Median function yielded the best estimates as it is more stable to outliers (we do not detail these experiments due to limited space).

5.3.2 Early Stopping. The described cardinality estimation technique considers all the matching queries to the given input query on the queries pool. However, we can configure the technique for early stopping. That is, taking into account all the matching queries in the pool is not always necessary. We can set a limit on the number of matching queries that are used to estimate the input query cardinality, and thus obtain predictions faster by considering only a subset of the matching queries.

In the reported experiments we consider all the queries in the pool since the pool size is limited as described in Section 6.2.

6 CARDINALITY EVALUATION

We evaluate our proposed technique for estimating cardinality, with different test sets, while using the CRN model as defined in Section 3.2 for estimating containment rates.

We compare our cardinality estimates with those of the PostgreSQL version 11 cardinality estimation component [1], a simple and commonly used method for cardinality estimation. In addition, we compare our cardinality estimates with those of the MSCN model [22]. MSCN was shown to be superior to the best methods for estimating cardinalities such as Random Sampling (RS) [5, 37] and the state-of-the-art Index-Based Join Sampling (IBJS) [27].

In order to make a fair comparison between the CRN model and the MSCN model, we train the MSCN model with the *same* data that was used to train the CRN model. The CRN model takes two queries as input, whereas the MSCN model takes one query as input. Therefore, to even the playing field, we created the training dataset for the MSCN model as follows. For each pair of queries ($Q1, Q2$) used in training the CRN model, we added the following two input queries to the MSCN training set:

- $Q1 \cap Q2$, along with its actual cardinality.
- $Q1$, along with its actual cardinality.

Finally, we ensure that the training set includes only unique queries without repetition. This way, both models, MSCN and CRN, are trained with the *same* information. Note that comparing with the profiling and histograms-based PostgreSQL does not require generating training set.

We create the test workloads using the same queries generator used for creating the training set of the CRN and the MSCN models (described in Section 3.1.2), while skipping its last step. That is, we only run the first two steps of the generator. The third step creates query pairs which are irrelevant for the cardinality estimation task.

6.1 Evaluation Workloads

We evaluate our approach on the (challenging) IMDb dataset, using three different query workloads:

- crd_test1, a synthetic workload generated by the same queries generator that was used for creating the training data of the CRN model, as described in Section 3.1 (using a different random seed) with 450 unique queries, with zero to two joins.
- crd_test2, a synthetic workload generated by the same queries generator as the training data of the CRN model, as described in Section 3.1 (using a different random seed) with 450 unique queries, with zero to *five* joins. This dataset is designed to examine how the technique generalizes to additional joins.
- scale, another synthetic workload, with 500 unique queries, derived from the MSCN test set as introduced in [22]. This dataset is designed to examine how the technique generalizes to queries that were *not* created with the same queries generator used for training.

number of joins	0	1	2	3	4	5	overall
crd_test1	150	150	150	0	0	0	450
crd_test2	75	75	75	75	75	75	450
scale	115	115	107	88	75	0	500

Table 3: Distribution of joins.

6.2 Queries Pool

Our technique relies on a queries pool, we thus created a synthetic queries pool, QP, generated by the same queries generator as the training data of the containment rate estimation model, as described in Section 3.1 (using a different random seed) with 300 queries, equally distributed among all the possible FROM clauses over the database. In particular, QP, covers all the possible FROM clauses that are used in the test workloads. Note that, there are no shared queries between the QP queries and the test workloads queries.

Consider a query Q whose cardinality needs to be estimated. On the one hand, the generated QP contains "similar" queries to query Q. These can help the machine in predicting the cardinality. On the other hand, it also includes queries that are not similar at all to query Q. These may cause erroneous cardinality estimates. Therefore, the generated queries pool QP, faithfully represents a real-world situation.

6.3 Experimental Environment

In all the following cardinality estimation experiments, for predicting the cardinality of a given query Q in a workload W, we use the whole queries pool QP as described in Section 6.2 with all its 300 queries. That is, the "old" queries used for predicting cardinalities, are the queries of QP. In addition, in all the experiments we use the Median function as the final F function.

6.4 The Quality of Estimates

Figure 7 depicts the q-error of the Cnt2Crd(CRN) model as compared to MSCN and PostgreSQL on the crd_test1 workload. While PostgreSQL's errors are more skewed towards the positive domain, MSCN is competitive with Cnt2Crd(CRN) in all the described values. As can be seen in Table 4, while MSCN provides the best results in the margins, the Cnt2Crd(CRN) model is more accurate in 75% of the tests (as it is less accurate, in the margins, than MSCN with queries that have up to two joins). In addition, we show in the next section (Section 6.5) that the Cnt2Crd(CRN) model is more robust when considering queries with more joins than in the training dataset.

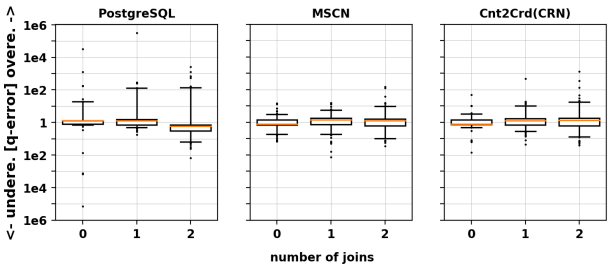

Figure 7: Estimation errors on the crd_test1 workload. In all the similar plots presented in this paper, the box boundaries are at the 25th/75th percentiles and the horizontal lines mark the 5th/95th percentiles. Hence, 50% of the tests results are located within the box boundaries, and 90% are located between the horizontal lines. The orange horizontal line mark the 50th percentile.

	50th	75th	90th	95th	99th	max	mean
PostgreSQL	1.74	3.72	22.46	149	1372	499266	1623
MSCN	2.11	4.13	**7.79**	**12.24**	**51.04**	**184**	**4.66**
Cnt2Crd(CRN)	**1.83**	**3.71**	10.01	18.16	76.54	1106	9.63

Table 4: Estimation errors on the crd_test1 workload.

6.5 Generalizing to Additional Joins

We examine how our technique generalizes to queries with additional joins, without having seen such queries during training. To do so, we use the crd_test2 workload which includes queries with zero to *five* joins. Recall that we trained both the CRN model and the MSCN model only with query pairs that have between zero and two joins.

From Tables 5 and 6, and Figure 8, it is clear that Cnt2Crd(CRN) model is significantly more robust in generalizing to queries with additional joins. This is clearly illustrated in the Cnt2Crd(CRN) box plot. The boxes are almost within the same q-error interval, close to q-error 1, which is the best q-error (obtained when an estimate is 100% accurate). In terms of mean q-error, the Cnt2Crd(CRN) model reduces the mean by a factor x100 and x1000 compared with MSCN and PostgreSQL, respectively.

	50th	75th	90th	95th	99th	max	mean
PostgreSQL	9.22	289	5189	21202	576147	4573136	35169
MSCN	4.49	119	3018	6880	61479	388328	3402
Cnt2Crd(CRN)	**2.66**	**6.50**	**18.72**	**72.74**	**528**	**6004**	**34.42**

Table 5: Estimation errors on the crd_test2 workload.

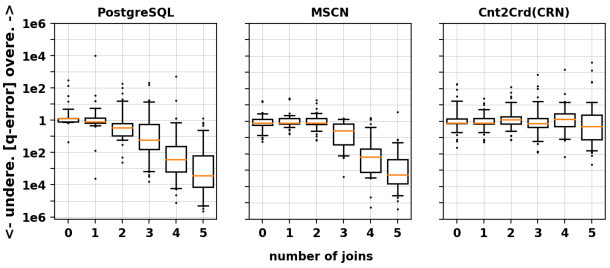

Figure 8: Estimation errors on the crd_test2 workload.

	50th	75th	90th	95th	99th	max	mean
PostgreSQL	229	3326	22249	166118	2069214	4573136	70569
MSCN	121	1810	6900	25884	83809	388328	6801
Cnt2Crd(CRN)	**4.28**	**10.84**	**43.71**	**93.11**	**1103**	**6004**	**61.26**

Table 6: Estimation errors on the crd_test2 workload considering only queries with three to five joins.

To highlight these improvements, we describe, in Table 7 and Figure 9, the mean and median q-error for each possible number of joins separately (note the logarithmic y-axis scale in Figure 9).

The known cardinality estimation models suffer from producing under-estimated results and errors that grow exponentially as the number of joins increases [14]. This also happens in the cases we examined. The Cnt2Crd(CRN) model was better at handling additional joins (even though CRN was trained only with queries with up to two joins, as was MSCN). The reason why the Cnt2Crd(CRN) model successfully generalizes to additional joins lies in its use of the queries pool. The queries pool contains queries with a *similar number of joins* as the input queries, along with their true cardinalities. The underlying CRN model estimates the containment rates accurately even when considering a high number of joins. As a result, the Cnt2Crd(CRN) cardinality estimates are accurate as well.

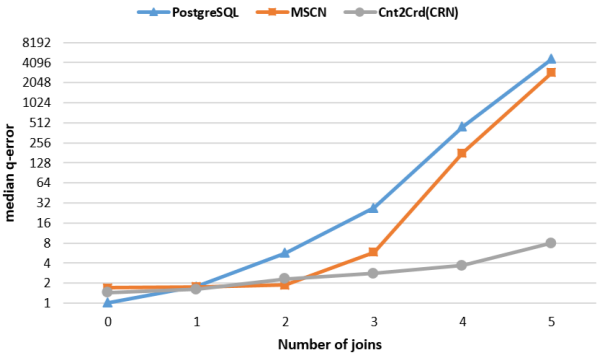

Figure 9: Q-error medians for each number of joins.

number of joins	0	1	2	3	4	5
PostgreSQL	10.41	216	25.38	355	4430	210657
MSCN	**3.44**	3.56	**3.31**	81.95	5427	14895
Cnt2Crd(CRN)	12.43	**3.54**	6.77	**23.24**	**30.51**	**129**

Table 7: Q-error means for each number of joins.

6.6 Generalizing to Different Kinds of Queries

In this experiment, we explore how the Cnt2Crd(CRN) model generalizes to a workload that was not generated by the same queries generator that was used for creating the CRN model training set. To do so, we examine the *scale workload* that is generated using another queries generator in [22]. As shown in Table 8, Cnt2Crd(CRN) is clearly more robust than MSCN and PostgreSQL in all the described percentiles. Examining Figure 10, it is clear that the Cnt2Crd(CRN) model is significantly more robust with queries with 3 and 4 joins. Recall that the *QP* queries pool in this experiment was not changed, while the scale workload is derived from *another* queries generator. In summary, this experiment shows that Cnt2Crd(CRN) generalizes well to workloads that were created with a different generator than the one used to create the training data.

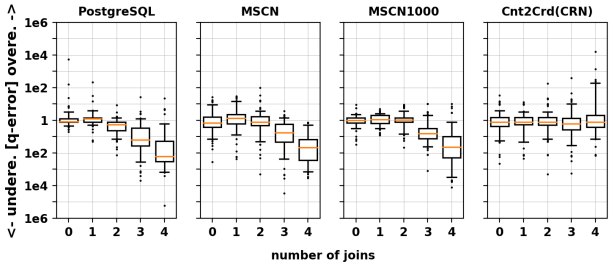

Figure 10: Estimation errors on the scale workload.

	50th	75th	90th	95th	99th	max	mean
PostgreSQL	2.62	15.42	183	551	2069	233863	586
MSCN	3.76	16.84	100	448	3467	47847	204
Cnt2Crd(CRN)	**2.53**	**5.88**	**24.02**	**95.26**	**598**	**19632**	**69.85**

Table 8: Estimation errors on the scale workload.

To further examine how Cnt2Crd(CRN) generalizes, we conducted the following experiment. We compared the Cnt2Crd(CRN) model with an improved version of the MSCN model that combines the deep learning approach and sampling techniques by using samples of 1000 materialized base tables, as described in [22]. We denote this model as MSCN1000.

We make the test "easier" for MSCN1000 model by training the MSCN1000 model with a training set that was created with the *same* queries generator that was used for generating the scale workload. As depicted in Figure 10, the MSCN1000 model is more robust in queries with zero to two joins, still, the Cnt2Crd(CRN) model is superior on queries with additional joins. Recall that the CRN model training set *was not changed*, while the MSCN1000 model was trained with queries obtained from the *same* queries generator that was used for creating the test (i.e., scale) workload. In addition, note that the MSCN1000 model uses sampling techniques whereas Cnt2Crd(CRN) does not. Thus, this experiment further demonstrates the superiority of Cnt2Crd(CRN) in generalizing to additional joins.

We obtain these improvements for the same reason described in Section 6.5. The CRN model is more robust in generalizing for additional unseen (during training) joins. As a result, the Cnt2Crd(CRN) model generalizes well for cardinality estimation.

7 IMPROVING EXISTING CARDINALITY ESTIMATION MODELS

In this section we describe how existing cardinality estimation models can be improved using the idea underlining our proposed technique. The proposed technique for improving existing cardinality estimation models relies on the same technique for predicting cardinalities using a containment rate estimation model, as described in Section 5.3.

In the previous section we used the CRN model in predicting containment rates. CRN can be replaced with *any* other method for predicting containment rates. In particular, it can be replaced with any existing cardinality estimation model after "converting" it to estimating containment rates using the Crd2Cnt transformation, as described in Section 4.1.

At first glance, our proposed technique seems to be a more complicated method for solving the problem of estimating cardinalities. However, we show that by applying it to known existing models, we improve their estimates, without changing the models themselves. These results indicate that the traditional approach, which directly addressed this problem, straightforwardly, using models to predict cardinalities, can be improved upon.

In the remainder of this section, we described the proposed approach, and show how existing cardinality estimation methods are significantly improved upon, by using this technique.

7.1 Approach Demonstration

Given an existing cardinality estimation model M, we first convert M to a model M' for estimating containment rates, using the Crd2Cnt transformation, as described in Section 4.1. Afterwards, given the obtained containment rate estimation model M', we convert it to a model M'' for estimating cardinalities, using the Cnt2Crd transformation, as described in Section 5.3, which uses a queries pool.

To summarize, our technique converts an existing cardinality estimation model M to an intermediate model M' for estimating containment rates, and then, using M' we create a model M'' for estimating cardinalities with the help of the queries pool, as depicted in Figure 5 from left to right.

For clarity, given cardinality estimation model M, we denote the model M'' described above, i.e., model Cnt2Crd(Crd2Cnt(M)), as *Improved* M model.

7.2 Existing Models vs. Improved Models

We examine how our proposed technique improves the PostgreSQL and the MSCN models, by using the crd_test2 workload as defined in Section 6.1, as it includes the most number of joins. Table 9 depicts the estimates when using directly the PostgreSQL or MSCN models, compared with the estimates when adopting our technique with each one of these models (i.e., the Improved PostgreSQL model and the Improved MSCN model). Examining the results, it is clear that the proposed technique significantly improves the estimates (by a factor x7 for PostgreSQL and x122 for MSCN in terms of mean q-error) without changing the models themselves (embedded within the Improved version).

The reason why the existing cardinality estimation models obtain better estimates (when adopting our technique) stems from the fact that these models are generalizing better when they are converted to estimate containment rates. Thus, along with the use of the queries pool, when these models are converted back to estimate cardinalities, they obtain better estimates.

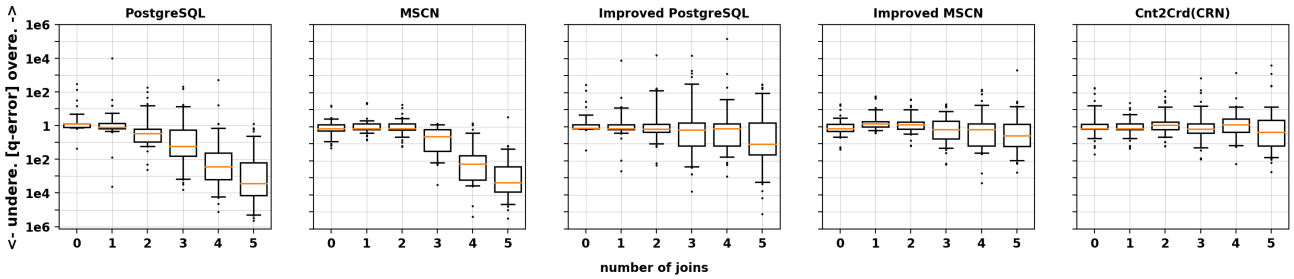

Figure 11: Estimation errors on the crd_test2 workload, compared with all models.

These results highlight the power of our proposed approach. The approach provides an effective and simple technique for improving existing cardinality estimation models. By adopting our approach and crating a queries pool in the database, cardinality estimates can be improved significantly.

	50th	75th	90th	95th	99th	max	mean
PostgreSQL	9.22	289	5189	21202	576147	4573136	35169
Improved PostgreSQL	**2.61**	**19.3**	**155**	**538**	**17697**	**1892732**	**5081**
MSCN	4.49	119	3018	6880	61479	388328	3402
Improved MSCN	**2.89**	**7.43**	**25.26**	**55.73**	**196**	**3184**	**27.78**

Table 9: Estimation errors on the crd_test2 workload.

7.3 Improved Models vs. Cnt2Crd(CRN)

Using the crd_test2 workloade, we examine how our technique improves PostgreSQL and MSCN, compared with Cnt2Crd(CRN). Examining Table 10, it is clear that in 90% of the tests, the best estimates are those obtained when directly using the CRN model to estimate the containment rates, instead of converting existing cardinality estimation models to obtain containment rates (Improved MSCN and Improved PostgreSQL). It seems that the CRN model is more accurate in estimating containment rates since it is directly designed for performing this task, whereas existing cardinality estimation models need to first be converted in order to estimate containment rates using the Crd2Cnt transformation.

	50th	75th	90th	95th	99th	max	mean
Improved PostgreSQL	2.61	19.3	155	538	17697	1892732	5081
Improved MSCN	2.89	7.43	25.26	**55.73**	**196**	**3184**	**27.78**
Cnt2Crd(CRN)	**2.66**	**6.50**	**18.72**	72.74	528	6004	34.42

Table 10: Estimation errors on the crd_test2 workload.

8 CARDINALITY PREDICTION TIME

Using the idea of using containment rates estimations to predict cardinalities, the cardinality prediction process is dominated by calculating the containment rates of the given input query with the relevant queries in the queries pool, and calculating the final function F on these results to obtain the predicted cardinality, as described in Section 5.3. Therefore, the larger the queries pool is, the more accurate the predictions are, and the longer the prediction time is. Table 11, shows the medians and the means estimation errors on the crd_test2 workload, along with the average prediction time for a single query, when using the Cnt2Crd(CRN) model for estimating cardinalities, with different sizes of QP (equally distributed over all the possible FROM clauses in the database) while using the same final function F (the Median function).

QP Size	50	100	150	200	250	300
Median	3.68	2.55	2.63	2.55	2.61	2.66
Mean	1894	90	41	40	35	34
Prediction Time	3.2ms	7.1ms	9.8ms	11.3ms	14.5ms	16.1ms

Table 11: Median and mean estimation errors on the crd_test2 workload, and the average prediction time, considering different queries pool (QP) sizes.

In table 12, we compare the average prediction time for estimating the cardinality of a single query using all the examined models (when using the whole QP queries pool of size 300). The default MSCN model is the fastest model, since it directly estimates the cardinalities without using a queries pool. The Cnt2Crd(CRN) model is the fastest among all the models that use a queries pool. That is, the Cnt2Crd(CRN) model is faster than the Improved MSCN model and the Improved PostgreSQL model. This is the case, since in the Improved MSCN model or the Improved PostgreSQL model, to obtain the containment rates, both models need to estimate cardinalities of two different queries as described in Section 4.1, whereas the CRN model directly obtains a containment rate in one pass within 0.5ms (see Section 3.5).

Although the prediction time of the models that use queries pools is higher than the most common cardinality estimation model (PostgreSQL), the prediction time is still in the order of a few tens milliseconds. In particular, it is similar to the average prediction time of models that use sampling techniques, such as the MSCN version with 1000 base tables samples.

For the results in Table 12, we used a queries pool (QP) of size 300. We could have used a smaller pool (or adapt the early stopping technique as mentioned in Section 5.3.2), resulting in faster prediction time, and still obtaining better results, as depicted in Table 11. Furthermore, all the the models that use queries pools may be easily parallelized as discussed in Section 5.3, and thus, reducing the prediction time (we ran these models serially in the reported tests).

Model	Prediction Time
PostgreSQL	1.75ms
MSCN	0.5ms
MSCN with 1000 samples	33ms
Improved PostgreSQL	70ms
Improved MSCN	35ms
Cnt2Crd(CRN)	16ms

Table 12: Average prediction time of a single query.

9 RELATED WORK

Over the past five decades, conjunctive queries have been studied in the contexts of database theory and database systems. Conjunctive queries constitute a broad class of frequently used queries. Their expressive power is roughly equivalent to that of the Select-Join-Project queries of relational algebra. Numerous problems and associated algorithms have been researched in depth in this context. Chandra and Merlin [10] showed that determining (analytic) containment of conjunctive queries is an NP-complete problem. Finding the minimal number of conditions that need to be added to a query in order to ensure containment in another query is also an NP-complete problem [44]. This also holds in additional settings involving inclusion and functional dependencies [2, 19, 44].

Although determining whether query $Q1$ is contained in query $Q2$ (analytically) in the case of conjunctive queries is an intractable problem in its full generality, there are many tractable cases. For instance, in [41, 42] it was shown that query containment of conjunctive queries could be solved in linear time, if every database (edb) predicate occurs at most twice in the body of $Q1$. In [12] it was proved that for every $k \geq 1$, conjunctive query containment could be solved in polynomial time, if $Q2$ has querywidth smaller than $k + 1$. In addition to the mentioned cases, there are many other tractable cases [8, 9, 16, 40]. Such cases result from imposing syntactic or structural restrictions on the input queries $Q1$ and $Q2$.

Whereas analytic containment was well researched in the past, to our knowledge, the problem of determining the containment rate on a *specific* database has not been investigated. In this paper, we address this problem using ML techniques.

Lately, we have witnessed extensive adoption of machine learning, and deep neural networks in particular, in many different areas and systems, and in particular in databases. Recent research investigates machine learning for classical database problems such as join ordering [31], index structures [23], query optimization [24, 38], concurrency control [4], and recently in cardinality estimation [22, 45]. MSCN, a recently conceived sophisticated NN model, estimates cardinalities [22]. MSCN has been shown to be superior in estimating cardinalities for queries that have the same number of joins as that in the queries training dataset. However, MSCN proved less effective when considering queries with more joins. In this paper, we propose a deep learning-based approach, inspired by the MSCN model, for predicting containment rates on a *specific* database. Additionally, we show how containment rates can be used to predict cardinalities more accurately.

There were many attempts to tackle the problem of cardinality estimation; for example, Random Sampling techniques [5, 37], Index based Sampling [27], and recently deep learning [22, 45]. However, all these attempts have addressed, conceptually, the problem directly in the same way, as a black box, where the input is a query, and the output is the cardinality estimate. In this paper, we address this problem differently by using information (the actual cardinalities) about queries that have already been executed in the database.

A similar idea of using the information contained in the execution results of queries was used to refine and update columns of histograms. In this approach, histograms are incrementally refined every time they are used, by comparing the histogram estimated selectivity to the actual selectivity. This leads to more accurate histograms, and to better cardinality estimates [3, 7, 13, 20].

10 CONCLUSIONS AND FUTURE WORK

We introduced a new problem, that of estimating containment rates between queries over a *specific* database, and introduced the CRN model, a new deep learning model for solving it (inspired by MSCN [22]). We trained CRN with generated queries, uniformly distributed within a constrained space, and showed that CRN usually obtains the best results in estimating containment rates as compared with other examined models.

We introduced a novel approach for cardinality estimation, based on the CRN-based containment rate estimation model, and with the help of a queries pool. We showed the superiority of our new approach in estimating cardinalities more accurately than state-of-the-art approaches. Further, we showed that our approach addresses the weak spot of existing cardinality estimation models, which is handling multiple joins.

In addition, we proposed a technique for improving *any* existing cardinality estimation model (M) without the need to change the model itself, by embedding it within a three step method ($Cnt2Crd(Crd2Cnt(M))$). Observe that it is possible to further improve the estimation by using the obtained improved model $Cnt2Crd(Crd2Cnt(M))$, and generating models (repeatedly), e.g., $Cnt2Crd(Crd2Cnt(Cnt2Crd(Crd2Cnt(M))))$[6]. Given that the estimates of state-of-the-art models are quite fragile, and that our technique for estimating cardinalities is simple, has low overhead, and is quite effective, we believe that it is highly promising and practical for solving the cardinality estimation problem.

We considered cardinality estimation for SQL queries *not* using the *DISTINCT* keyword. For various intermediate results, a query planner requires the set-theoretic cardinality (without duplicates). For example, employing counting techniques for handling duplicates, considering sorting, creating an index or a hash table, and more. This requirement may therefore limit our techniques' usability. One may use our (inaccurate for this case) predictions as proxies. However, a better technique is needed and we are currently evaluating a promising extension of our machine learning approach for predicting set-theoretic cardinalities (i.e., queries with the *DISTINCT* keyword).

To make our containment based approach suitable for more general queries, the CRN model for estimating containment rates can be extended to support other types of queries, such as queries that include complex predicates. In addition, the CRN model can be configured to support databases that are updated from time to time. Next, we discuss some of these extensions, and sketch possible future research directions.

Strings. A simple addition to our current implementation may support equality predicates on strings. To do so, we could hash all the possible string literals in the database into the integer domain (similarly to MSCN). This way, an equality predicate on strings can be converted to an equality predicate on integers, which the CRN model can handle.

Complex predicates. Complex predicates, such as LIKE, are not supported since they are not represented in the CRN model. To support such predicates we need to change the model architecture to handle such predicates. Note that predicates such as BETWEEN and IN, may be converted to ordinary predicates.

EXCEPT Operator. Given a query Q of the form $Q1$ EXCEPT $Q2$, we can estimate its cardinality using our technique as follows:

$$|Q1\ EXCEPT\ Q2| = |Q1| - |Q1 \cap Q2|$$

[6] This observation is due to one of the referee.

UNION Operator. Given a query Q of the from $Q1$ UNION $Q2$, we can estimate its cardinality using our technique as follows:

$$|Q1\ UNION\ Q2| = |Q1| + |Q2|$$

Observe that for handling both the EXCEPT and the UNION operators, the cardinality of queries $Q1$, $Q2$ and $Q1 \cap Q2$ can be estimated using our technique, as they are conjunctive queries.

The OR operator. Given queries that include the OR operator in their WHERE clause, the CRN model does not handle such queries straightforwardly. But, we can handle such queries using a promising recursive algorithm that we are currently evaluating.

Database updates. Thus far, we assumed that the database is static (read-only database). However, in many real world databases, updates occur frequently. In addition, the database schema itself may be changed. To handle updates we can use one of the following approaches:

(1) We can always completely re-train the CRN model with a new updated training set. This comes with a considerable compute cost for re-executing queries pairs to obtain up-to-date containment rates and the cost for re-training the model itself. In this approach, we can easily handle changes in the database schema, since we can change the model encodings prior to re-training it.

(2) We can *incrementally* train the model starting from its current state, by applying new updated training samples, instead of re-training the model from scratch. While this approach is more practical, a key challenge here is to accommodate changes in the database schema. To handle this issue, we could hold, in advance, additional place holders in our model to be used for future added columns or tables. In addition, the values ranges of each column may change when updating the database, and thus, the normalized values may be modified as well. Ways to handle this problem are the subject of current research.

REFERENCES

[1] . .. PostgreSQL, The World's Most Advanced Open Source Relational Database. *https://www.postgresql.org/* (.).

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison-Wesley.

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

[4] Rajesh Bordawekar and Oded Shmueli (Eds.). 2019. *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD.* ACM.

[5] Marco Bressan, Enoch Peserico, and Luca Pretto. 2015. Simple set cardinality estimation through random sampling. *CoRR* abs/1512.07901 (2015).

[6] Jason Brownlee. 2018. When to Use MLP, CNN, and RNN Neural Networks. *https://machinelearningmastery.com/when-to-use-mlp-cnn-and-rnn-neural-networks/* (2018).

[7] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. 2001. STHoles: A Multidimensional Workload-Aware Histogram. In *Proceedings SIGMOD.* 211–222.

[8] Andrea Calì. 2006. Containment the Conjunctive Queries over Conceptual Schemata. In *Proceedings the Database Systems for Advanced Applications conference,DASFAA.* 628–643.

[9] Edward P. F. Chan. 1992. Containment and Minimization of Positive Conjunctive Queries in OODB's. In *Proceedings SIGACT-SIGMOD-SIGART.* 202–211.

[10] Ashok K. Chandra and Philip M. Merlin. 1977. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing.* 77–90.

[11] Gloria Chatzopoulou, Magdalini Eirinaki, and Neoklis Polyzotis. 2009. Query Recommendations for Interactive Database Exploration. In *Scientific and Statistical Database Management, 21st International Conference, SSDBM.* 3–18.

[12] Chandra Chekuri and Anand Rajaraman. 2000. Conjunctive query containment revisited. *Theor. Comput. Sci.* 239, 2 (2000), 211–229.

[13] Chung-Min Chen and Nick Roussopoulos. 1994. Adaptive Selectivity Estimation Using Query Feedback. In *Proceedings ACM SIGMOD.* 161–172.

[14] James Clifford and Roger King (Eds.). 1991. *Proceedings the 1991 ACM SIGMOD International Conference on Management of Data.* ACM Press.

[15] Magdalini Eirinaki, Suju Abraham, Neoklis Polyzotis, and Naushin Shaikh. 2014. QueRIE: Collaborative Database Exploration. *IEEE Trans. Knowl. Data Eng.* 26, 7 (2014), 1778–1790.

[16] Carles Farré, Werner Nutt, Ernest Teniente, and Toni Urpí. 2007. Containment of Conjunctive Queries over Databases with Null Values. In *Proceedings ICDT.* 389–403.

[17] Archana Ganapathi, Harumi A. Kuno, Umeshwar Dayal, Janet L. Wiener, Armando Fox, Michael I. Jordan, and David A. Patterson. 2009. Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning. In *Proceedings ICDE.* 592–603.

[18] Rojeh Hayek and Oded Shmueli. 2019. Improved Cardinality Estimation by Learning Queries Containment Rates. *CoRR* abs/1908.07723 (2019). arXiv:1908.07723 http://arxiv.org/abs/1908.07723

[19] David S. Johnson and Anthony C. Klug. 1984. Testing Containment of Conjunctive Queries under Functional and Inclusion Dependencies. *J. Comput. Syst. Sci.* 28, 1 (1984), 167–189.

[20] Navin Kabra and David J. DeWitt. 1998. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In *SIGMOD.* 106–117.

[21] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *Proceedings ICLR.* http://arxiv.org/abs/1412.6980

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

[23] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings SIGMOD.* 489–504.

[24] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. 2018. Learning to Optimize Join Queries With Deep Reinforcement Learning. *CoRR* abs/1808.03196 (2018).

[25] M. Seetha Lakshmi and Shaoyu Zhou. 1998. Selectivity Estimation in Extensible Databases - A Neural Network Approach. In *Proceedings VLDB.* 623–627.

[26] 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.

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

[28] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *VLDB J.* 27, 5 (2018), 643–668.

[29] Henry Liu, Mingbin Xu, Ziting Yu, Vincent Corvinelli, and Calisto Zuzarte. 2015. Cardinality estimation using neural networks. In *Proceedings the 25th Annual International Conference on Computer Science and Software Engineering,CASCON.* 53–59.

[30] Guy Lohman. 2014. IS QUERY OPTIMIZATION A "SOLVED" PROBLEM ? *https://wp.sigmod.org/?p=1075* (2014).

[31] Ryan Marcus and Olga Papaemmanouil. 2018. Deep Reinforcement Learning for Join Order Enumeration. In *Proceedings aiDM@SIGMOD.* 3:1–3:4.

[32] Alberto O. Mendelzon and Jan Paredaens (Eds.). 1998. *Proceedings the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems.* ACM Press.

[33] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors. *PVLDB* 2, 1 (2009), 982–993.

[34] R. Monga, M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *OSDI 16.* 265–283.

[35] Raghunath Nambiar and Meikel Poess (Eds.). 2018. *Performance Evaluation and Benchmarking for the Analytics Era - 9th TPC Technology Conference, TPCTC.* Lecture Notes in Computer Science, Vol. 10661. Springer.

[36] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. 2018. Activation Functions: Comparison of trends in Practice and Research for Deep Learning. *CoRR* abs/1811.03378 (2018).

[37] Frank Olken and Doron Rotem. 1990. Random Sampling from Database Files: A Survey. In *Proccedings SSDBM.* 92–111.

[38] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. 2018. Learning State Representations for Query Optimization with Deep Reinforcement Learning. In *Proceedings, DEEM@SIGMOD.* 4:1–4:4.

[39] Lutz Prechelt. 2012. Early Stopping - But When? In *Neural Networks: Tricks of the Trade - Second Edition.* 53–67.

[40] Guillem Rull, Philip A. Bernstein, Ivo Garcia dos Santos, Yannis Katsis, Sergey Melnik, and Ernest Teniente. 2013. Query containment in entity SQL. In *Proceedings ACM SIGMOD.* 1169–1172.

[41] Y. Saraiya. 1991. Subtree elimination algorithms in deductive databases. *PhD thesis, Department of Computer Science, Stanford University.* (1991).

[42] Yatin P. Saraiya. 1990. Polynomial-time Program Transformations in Deductive Databases. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '90).* ACM, 132–144.

[43] Cristina Sirangelo. 2018. Positive Relational Algebra. In *Encyclopedia of Database Systems, Second Edition.* Computer Science Press.

[44] Jeffrey D. Ullman. 1989. *Principles the Database and Knowledge-Base Systems, Volume II.* Computer Science Press.

[45] Lucas Woltmann, Claudio Hartmann, Maik Thiele, Dirk Habich, and Wolfgang Lehner. 2019. Cardinality estimation with local deep learning models. In *Proceedings aiDM@SIGMOD.* 5:1–5:8.