Machine Learning of SQL Queries Containment Rate and Result Cardinality

Rojeh Hayek
Machine Learning of SQL Queries Containment Rate and Result Cardinality

Research Thesis

Submitted in Partial Fulfillment of The Requirements for the degree of Master of Science in Computer Science

Rojeh Hayek

Submitted to the Senate of the Technion - Israel Institute of Technology
Cheshvan 5780 Haifa November 2019
This research was carried out under the supervision of Prof. Oded Shmueli, in the Faculty of Computer Science.

Some results in this thesis have been published as articles by the author (Rojeh Hayek) and the advisor (Oded Shmueli) during the course of this master research, the most up-to-date versions of which are:

<table>
<thead>
<tr>
<th>Title</th>
<th>Journal/Conference</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Improved cardinality estimation by learning queries containment rates</td>
<td>CoRR, abs/1908.07723</td>
<td>2019</td>
</tr>
<tr>
<td>Improved cardinality estimation by learning queries containment rates</td>
<td>To appear in Advances in Database Technology - 23nd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020</td>
<td>2020</td>
</tr>
</tbody>
</table>

**Acknowledgements**

Firstly, I would like to express my sincere gratitude to my advisor Prof. Oded Shmueli for the continuous support of my master study, for his patience, motivation, and immense knowledge. The door to his office was always open whenever I ran into a trouble spot or had a question about my research or writing. It was an honor and privilege to work with you Prof. Shmueli.

Besides, I wish to give my very profound gratitude to all my friends, especially, Mira Haddad. I really appreciate all the time you have dedicated listening to me, and providing me with unfailing support and continuous encouragement throughout my studies and the process of researching.

My acknowledgement would be incomplete without thanking the biggest source of my strength, my parents, my brother and my sisters. The unconditional love and support that they gave me throughout my studies and my life in general are invaluable.

The generous financial help of the Technion is gratefully acknowledged.
Contents

List of Figures

Abstract 1

1 Introduction 3

2 Containment Rates 7
  2.1 Containment Rate Definition .......................... 7
    2.1.1 Containment Rate Operator ...................... 8
  2.2 Learned Containment Rates .......................... 9
    2.2.1 Cold Start Problem .............................. 10
    2.2.2 Model ........................................ 14
    2.2.3 A Simple Example .............................. 18
    2.2.4 Training and Testing Interface ................. 19
    2.2.5 Hyperparameter Search .......................... 20
    2.2.6 Model Computational Costs .................... 21
  2.3 CRN Evaluation ..................................... 22
    2.3.1 From Cardinality to Containment ............... 22
    2.3.2 Evaluation Workloads ........................... 24
    2.3.3 Evaluation Results Representations ............ 24
    2.3.4 The Quality of Estimates ..................... 25
    2.3.5 Generalizing to Additional Joins .............. 26

3 Cardinality Estimation Using Containment Rates 27
  3.1 Introduction ....................................... 27
    3.1.1 From Containment to Cardinality ............... 28
    3.1.2 Query Pool .................................... 30
    3.1.3 A Cardinality Estimation Technique ............ 31
  3.2 Cnt2Crd Evaluation ................................. 33
    3.2.1 Evaluation Workloads ........................... 33
3.2.2 Query Pool ........................................... 34
3.2.3 Experimental Environment ............................ 34
3.2.4 The Quality of Estimates ............................ 35
3.2.5 Generalizing to Additional Joins ..................... 36
3.2.6 Generalizing to Different Kinds of Queries .......... 38
3.3 Improving Existing Cardinality Estimation Models ...... 40
  3.3.1 Approach Demonstration ............................ 40
  3.3.2 Existing Models vs. Improved Models ................ 42
  3.3.3 Improved Models vs. Cnt2Crd(CRN) ................. 43
3.4 Cardinality Prediction Time ............................ 45

4 Estimating Set-Theoretic Cardinality .................. 47
  4.1 Introduction ........................................ 47
  4.2 Uniqueness Rate Definition ........................... 48
  4.3 Learned Uniqueness Rates ............................ 49
  4.3.1 Cold Start Problem ............................... 50
  4.3.2 Model ........................................... 51
  4.3.3 Training and Testing Interface ..................... 53
  4.3.4 Hyperparameter Search and Model Costs .......... 54
  4.3.5 Model Computational Costs ........................ 54
  4.4 PUNQ Evaluation .................................... 55
  4.4.1 Evaluation Workloads ............................. 55
  4.4.2 The Quality of Estimates .......................... 56
  4.4.3 Generalizing to Additional Joins .................. 58
  4.5 Cardinality Prediction Time .......................... 60

5 Supporting AND, OR and NOT operators ................. 61
  5.1 Introduction ........................................ 61
  5.2 The GenCrd Algorithm ................................ 62
  5.2.1 First Observation ................................ 62
  5.2.2 Second Observation ............................... 62
  5.2.3 The ImplyFalse Algorithm ........................ 64
  5.2.4 A simple example ................................ 66
  5.3 GenCrd Evaluation .................................. 67
  5.3.1 Evaluation Workloads ............................. 67
  5.3.2 The Quality of Estimates .......................... 68
  5.3.3 Generalizing to Additional Joins .................. 69
  5.4 Cardinality Prediction Time .......................... 70
List of Figures

2.1 Containment Rate Estimation Model as a Monolithic Box. . . . . 9
2.2 CRN Model Architecture. . . . . . . . . . . . . . . . . . . . . . . 14
2.3 The mean q-error on the validation set with different H sizes. . . 20
2.4 CRN model convergence of the mean q-error on the validation set. 21
2.5 Containment Rate Estimation Using A Cardinality Estimation Model. 23
2.6 Containment rate estimation q-errors on the cnt_test1 workload. . 25
2.7 Containment rate estimation q-errors on the cnt_test2 workload. . 26

3.1 Cardinality Estimation Model as a Monolithic Box. . . . . . . . . 28
3.2 Cardinality Estimation Using Containment Rate Estimation Model. 29
3.3 Cardinality Estimation Technique. . . . . . . . . . . . . . . . . . . 31
3.4 Cardinality estimation q-errors on the crd_test2 workload, while considering different Final (F) functions. . . . . . . . . . . . . . . . . . . . 32
3.5 Cardinality estimation q-errors on the crd_test1 workload. . . . . 35
3.6 Cardinality estimation q-errors on the crd_test2 workload. . . . . 36
3.7 Q-error medians for each number of joins. . . . . . . . . . . . . . . 38
3.8 Cardinality estimation q-errors on the scale workload. . . . . . . 39
3.9 A novel approach, from cardinality estimation to containment rate estimation, and back to cardinality estimation by using a query pool. 41
3.10 Cardinality estimation q-errors on the crd_test2 workload. Comparing the basic and the Improved PostgreSQL model. . . . . . . 42
3.11 Cardinality estimation q-errors on the crd_test2 workload. Comparing the basic and the Improved MSCN model. . . . . . . . . 43
3.12 Cardinality estimation q-errors on the crd_test2 workload. Comparing with all the examined models. . . . . . . . . . . . . . . . . . 44

4.1 Uniqueness-rate Estimation Model as a Monolithic Box. . . . . 49
4.2 PUNQ Model Architecture. . . . . . . . . . . . . . . . . . . . . . . . . 51
4.3 Uniqueness rates estimation q-errors on the UnqCrd_test1 workload. 56
4.4 Cardinality estimation q-errors on the UnqCrd_test1 workload. . . 57
4.5 Uniqueness rates estimation q-errors on the UnqCrd_test2 workload. 58
4.6 Cardinality estimation q-errors on the UnqCrd_test2 workload. 59

5.1 The GenCrd Algorithm. 63
5.2 The ImplyFalse Algorithm. 65
5.3 Cardinality estimation q-errors on the GenCrd_test1 workload. 68
5.4 Cardinality estimation q-errors on the GenCrd_test2 workload. 69

A.1 Box and Whisker Plot. 79
A.2 PostgreSQL cardinality estimates for different test workloads on the IMDb and the TPC-H databases. 80
A.3 A simple example, illustrating how the partial derivative of $E_{total}$ with respect to $w_1$ is calculated, using the chain rule ($E_{total}$ is the Loss function). 83
Abstract

A traditional query optimizer is crucially dependent on cardinality estimation of intermediate results, which enables choosing among different plan alternatives. Therefore, the query optimizer must use reasonably good estimates. However, estimates produced by all widely-used database cardinality estimation models (methods) are routinely significantly wrong, resulting in not choosing the best plans, leading to slow executions. In this work we define a new problem, that of estimating the containment rate between pairs of SQL queries (over a given database), and we introduce a novel technique for efficiently estimating queries’ result-cardinalities using estimated queries’ containment rates.

The containment rate of query $Q_1$ in query $Q_2$ over database $D$ is the percentage of $Q_1$’s result tuples over $D$ that are also in $Q_2$’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 which enables us to express query features using sets of vectors and vectors.

Experimentally, our novel approach for estimating cardinalities, using containment rates between queries, on a challenging real-world database, realizes significant improvements over state of the art cardinality estimation methods. We also show that by employing any existing cardinality estimation method $M$ (such as PostgreSQL), for containment rate estimation, and then embedding it in our cardinality estimation technique (replacing the CRN model), we improve on $M$’s cardinality estimates as well, without changing the method $M$ itself.

Additionally, we propose two techniques for extending any “limited” cardinality estimation model (that only estimates cardinalities with duplicates of conjunctive queries) to support general queries that use the AND, OR, and NOT operators, and to estimate set-theoretic cardinalities (i.e., cardinalities without duplicates). Experimentally, we show that when using these techniques, the extended models maintain the same quality of estimates as the original underlying models’ quality of estimates.
Chapter 1

Introduction

Containment Rate

Query \( Q_1 \) is contained in (resp. equivalent to), query \( Q_2 \), analytically, if for all database states \( D \), \( Q_1 \)’s result over \( D \) is contained in (resp., equals) \( Q_2 \)’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 [RBdS+13, Cal06, Cha92, FNTU07]. 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:

\( Q_1 \): select * from movies where title = ’Titanic’

\( Q_2 \): select * from movies where release = 1997 and director = ’James Cameron’

Since James Cameron directed only one movie in 1997, called Titanic, the execution results of both queries are identical. 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 \( Q_1 \) in query \( Q_2 \) on database \( D \) is the percentage of rows (tuples) in \( Q_1 \)’s execution result over \( D \) that are also in \( Q_2 \)’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. In this research we focus on their use for 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.

**Cardinality Estimation**

In addition to the CRN model, we introduce a novel technique Cnt2Crd($M$) for estimating queries’ cardinalities using query containment rates. The containment rates estimates can be obtained using any model $M$ for estimating containment rates, in particular using the CRN model. We show that by using the proposed technique with the CRN model for estimating containment rates (Cnt2Crd(CRN)), we improve over 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 [CK91]. 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 [LGM+15], to obtain an efficient query plan, the query optimizer chooses the cheapest alternative from semantically equivalent plan alternatives. Since the cost model uses 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 [DBM], and the pioneering multi-set convolutional network (MSCN) model [KKR+19], by examining, on the real-world IMDb database, join crossing correlations queries which are known to present a tough challenge to cardinality estimation methods [LGM+15, LRG+18, NP18].

We show that by converting an existing cardinality estimation method $M$ (e.g., MSCN) into one for estimating containment rates $M'$, we can improve on method
M’s cardinality estimates as well. This is done by simply using the converted model $M'$ in our Cnt2Crd technique (Cnt2Crd($M'$)). Thus, our novel approach is highly promising for solving the cardinality estimation problem, the "Achilles heel" of query optimization [Loh14], a cause of many performance issues [LGM+15].

**Extending Limited Cardinality Estimation Models**

Our proposed technique for estimating cardinalities using query containment rates, along with other recently developed cardinality estimation models such as the MSCN model, are limited to estimating cardinalities for conjunctive queries only. However, queries with the AND, OR, and NOT operators constitute a broad class of frequently used queries. Furthermore, the abovementioned models were designed to only estimate cardinalities with duplicates, thus further limiting the class of supported queries as queries with the DISTINCT keyword are simply not supported. In addition to not supporting queries with the DISTINCT keyword, for various intermediate results, a query planner requires the set-theoretic cardinality (without duplicates) even when the DISTINCT keyword is not used explicitly in the query. For example, in employing counting techniques for handling duplicates, in considering sorting options, and in creating an index or a hash table.

With all the improvements of the recently proposed models, the abovementioned limitations pose a significant problem to the applicability of these models. In the real world systems, supporting queries with the AND, OR, and NOT operators and the DISTINCT keyword is essential. The limited models implementations and techniques vary significantly. Thus, restructuring each one, separately, to support queries that are more general, is complex and potentially error-ridden. Therefore, to tackle this deficiency, we propose two separate methods for extending any limited model\(^1\), without the need to change the model itself.

First, we tackle the lack of support for the set-theoretic cardinality estimation (cardinalities without duplicates) using a specialized deep learning method, PUNQ. The PUNQ model is tailored to representing conjunctive SQL queries and predicting the percentage of unique rows within the query’s result with duplicates. Given a query $Q$ whose set-theoretic cardinality needs to be estimated, and a limited model $M$, the set-theoretic cardinality can be estimated as follows. Assuming that $U$ is the predicted percentage of unique rows in the query’s result obtained from PUNQ, and $C$ is the estimated cardinality with duplicates, obtained from model $M$, the value $C \cdot U$ is the estimated set-theoretic cardinality.

---

\(^1\)In this work, limited models refers to models that support only conjunctive queries and whose estimates count duplicates.
The PUNQ model uses a tailored vector representation for representing the input queries, similarly to the CRN model, which enables expressing the query’s features. As a result, we obtain a small and accurate model for estimating the percentage of unique rows in a query result that includes duplicates.

Queries that include the AND, OR, and NOT operators in their WHERE clauses constitute a broad class of frequently used queries. Their expressive power is equivalent to that of a large fragment of the relational algebra [Wikb] (in the case examined in this work, we consider only queries with equality joins). Therefore, estimating cardinalities for such queries is essential.

To estimate the result cardinality of such queries, we introduce a recursive algorithm, GenCrd, that estimates the cardinality of a general query (that includes the AND, OR, and NOT operators), using any limited model that only estimates cardinalities for conjunctive queries, as a "subroutine".

We evaluated our extension methods (both PUNQ and GenCrd), on the challenging, real-world IMDb database [LGM+15] with general queries that either include the DISTINCT keyword or the AND, OR, and NOT operators.

The extension methods were applied to our technique for estimating cardinality, in addition they were applied to the MSCN model. The extended methods were shown to produce cardinality estimates with the same level of accuracy as that of the original methods. This indicates the practicality of our extension methods, offering a simple and efficient tool for extending any limited cardinality estimation method, to support queries that are more general. We also compare the four extended methods with PostgreSQL [DBM].

Organization

The rest of this work is organized as follows. In Chapter 2 we define the containment rate problem, and describe and evaluate the CRN model for solving it. In Chapter 3 we describe and evaluate our new approach for estimating cardinalities using containment rates. Additionally, we show how one can adapt the new ideas to improve any existing cardinality estimation model. In Chapter 4 we describe and evaluate the PUNQ model for extending any limited model to support the DISTINCT keyword (i.e., estimating set-theoretic cardinalities). In Chapter 5, we describe and evaluate recursive algorithm GenCrd, that extends any limited model to support queries with the AND, OR, and NOT operators. Finally, in Chapter 6, we present related work, and in Chapter 7 we present our conclusions, and plans for future work.
Chapter 2

Containment Rates

2.1 Containment Rate Definition

We define the containment rate between two queries $Q_1$ and $Q_2$ on a specific database $D$. Query $Q_1$ is $x\%$-contained in query $Q_2$ on database $D$ if precisely $x\%$ of $Q_1$’s execution result rows on database $D$ are also in $Q_2$’s execution result on database $D$. The containment rate is formally a function from $Q \times Q \times D$ 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 $Q_1(D) = (A, m_{11})$ and $Q_2(D) = (B, m_{21})$ be multisets\(^1\) representing queries $Q_1$ and $Q_2$ execution results on database $D$, respectively, then:

$$x\% = \frac{\sum_{x \in (A \cap B)} m_{11}(x)}{\sum_{x \in A} m_{11}(x)} \times 100$$

Where operator $\cap$ is the regular set intersection operator (in case $Q_1$’s execution result is empty, then $Q_1$ is 0%-contained in $Q_2$). 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.

\(^1\)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 $m(x)$. 
2.1.1 Containment Rate Operator

We denote the containment rate *operator* between queries $Q_1$ and $Q_2$ on database $D$ as:

$$\text{Cnt}_{\%}(Q_1, Q_2, D)$$

Operator $\text{Cnt}_{\%}$ returns the containment rate between the given input queries on database $D$. That is, $\text{Cnt}_{\%}(Q_1, Q_2, D)$ returns $x\%$, if $Q_1$ is $x\%$-contained in query $Q_2$ on database $D$. For simplicity, we do not mention the *specific* database, as it is usually clear from context. Hence, we use the operator as follows $\text{Cnt}_{\%}(Q_1, Q_2)$.

*Example 2.1.1.* Consider database $D$, and the following SQL queries:

<table>
<thead>
<tr>
<th>From</th>
<th>Destination</th>
<th>duration</th>
<th>stops</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLV</td>
<td>Yyz</td>
<td>12:30</td>
<td>0</td>
</tr>
<tr>
<td>Yyz</td>
<td>TXL</td>
<td>08:30</td>
<td>0</td>
</tr>
<tr>
<td>JFK</td>
<td>TLV</td>
<td>12:50</td>
<td>2</td>
</tr>
<tr>
<td>JFK</td>
<td>TLV</td>
<td>10:00</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2.1: Database $D$.

Q1: SELECT * FROM D WHERE stops > 0  
Q2: SELECT * FROM D WHERE destination == TLV AND stops == 2  
Q3: SELECT * FROM D WHERE From = JFK  
Q4: SELECT * FROM D WHERE stops = 0

Queries Q1 and Q2 execution results on database $D$ are identical (the highlighted portion of $D$), in particular, they are 100\%-contained in each other, i.e.,

$$\text{Cnt}_{\%}(Q_1, Q_2) = 1, \, \text{Cnt}_{\%}(Q_2, Q_1) = 1$$

Query Q3 execution result on database $D$ are the last two rows, therefore, we get that:

$$\text{Cnt}_{\%}(Q_1, Q_3) = 1, \, \text{Cnt}_{\%}(Q_2, Q_3) = 1, \, \text{Cnt}_{\%}(Q_3, Q_1) = 0.5, \, \text{Cnt}_{\%}(Q_3, Q_2) = 0.5$$

Query Q4 execution result on database $D$ is the unlighted portion of $D$, therefore we get that:

$$\text{Cnt}_{\%}(Q_1, Q_4) = 0, \, \text{Cnt}_{\%}(Q_2, Q_4) = 0, \, \text{Cnt}_{\%}(Q_4, Q_1) = 0, \, \text{Cnt}_{\%}(Q_4, Q_2) = 0$$

$$\text{Cnt}_{\%}(Q_3, Q_4) = 0.5, \, \text{Cnt}_{\%}(Q_4, Q_3) = 0.33$$
2.2 Learned Containment Rates

We aim to use the containment rates, in solving the cardinality estimation problem, as described in Chapter 3. Therefore, calculating the actual containment rates of pairs of queries by definition, using their actual execution results, is impractical. The queries execution time may be too long, whereas, in the cardinality estimation problem, the estimated cardinalities need to be obtained within milliseconds. Thus, we propose a deep learning based model, denoted as Containment Rate Network (CRN), tailored for estimating containment rates between pairs of queries, without the need to actually execute them on the database.

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 \((Q_1, Q_2)\) and the actual containment rates \(\text{Cnt}_\%((Q_1, Q_2))\), the model is used as an estimator for other, unseen pairs of queries. (Later on, as described in Chapter 3, we will make use of this model to estimate cardinalities of single queries).

![Containment Rate Estimation Model as a Monolithic Box.](image)

There are, however, several questions whose answers determine whether the machine learning model (CRN) will be successful.

- Which supervised learning algorithm/model should be used.
- How to represent queries as input and the containment rates as output to the model ("featurization").
- How to obtain the initial training dataset ("cold start problem").

In the following sections, we describe how we address each one of these questions.
2.2.1 Cold Start Problem

Defining the Database

We generated a training-set, and later evaluated our model on it, using the IMDb database. IMDb contains many correlations and has been shown to be challenging for cardinality estimators [LGM+15] (see Appendix A.2). 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.

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 query generator generates the dataset in three main steps. In the first step, it generates multiple single queries. In the second step, it creates additional queries using the queries obtained from the first step, by randomly changing and adding columns predicates to the original queries. Finally, in the third and last step, using the queries obtained from both previous steps, it generates pairs of queries whose FROM clauses are identical.

The query generator includes three variables, $N$, $M$, and $tables-num-threshold$, that can be configured with different values for each run.

First Step:

For $t$ in range(1, $tables-num-threshold$+1): /* range(n,k) = [n,n+1,...,k-1] */

0. For each possible set of $t$ tables ($t = \{bt_1, bt_2, ..., bt_{|t|}\}$) that can join with each other in the database:

- For $i$ in range(0, $M$):
  - Add $|t| - 1$ 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 $bt$ from the $t$ chosen tables:
    * Uniformly draw the number of column predicates $p_{bt}$ of the current table $bt$ ($0 \leq |p_{bt}| \leq \text{num of columns in table } bt$).
* For j in range(0, \( p_t \)):
  · Uniformly draw a non-key column from the current table \( bt \).
  · Uniformly draw the predicate type (\(<\), \(=\), or \(>\)).
  · Uniformly draw a value from the corresponding column values interval in the database.

To avoid a combinatorial explosion, and to simplify the problem that the model needs to learn, we only generate queries with up to two joins and let the model generalize to a larger number of joins (i.e., we configure \( tables-num-threshold \) to 3). We also configured our query generator to only generate unique queries without repetitions of queries.

By running the first step, we obtain \( P \times M \) different queries, denoted as \( initial-queries \), where \( P \) is the number of possible sets of \( t \) tables that can join with each other in the database for all \( 1 \leq t \leq tables-num-threshold \). In practice, in the examined database, when \( t = 1 \) there are 6 possible sets. When \( t = 2 \) there are 5 possible sets. When \( t = 3 \) there are 10 possible sets. Therefore, given that \( tables-num-threshold = 3 \), we get that \( P = 5 + 6 + 10 = 21 \).

**Second Step:**
For each query \( Q \) in \( initial-queries \):

- For i in range(0, \( N/3 \)) create a new query as \( Q \) with the following changes:
  - Uniformly choose columns predicates from \( Q \)’s WHERE clause.
  - For each chosen column predicate, change the predicate value, by uniformly drawing a value from the corresponding column values interval in the database.

- For i in range(0, \( N/3 \)) create a new query as \( Q \) with the following changes:
  - Uniformly choose columns predicates from \( Q \)’s WHERE clause.
  - For each chosen column predicate, change the predicate operator, by uniformly drawing a new operator (\(<\), \(=\), or \(>\)).

- For i in range(0, \( N/3 \)) create a new query as \( Q \) with the following changes:
  - Uniformly choose columns predicates from \( Q \)’s WHERE clause.
  - For each chosen column predicate if the predicate type is \(=\), then skip this predicate. Otherwise, add an additional predicate on the same
column, with the opposite operator (i.e., if the original operator was $<$, then use $>$, and vice versa), and draw the new predicate value uniformly from the corresponding column values interval in the database.

In order 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, since they include different predicates values and/or operators. This way, we create a "hard" dataset, which includes pairs of queries that look "similar", but having mutual containment rates that vary significantly, as described in the third step.

By running the second step using the $P \times M$ initial-queries obtained from the first step, we obtain another $P \times M \times N$ different queries (for each query from the $P \times M$ initial-queries, we create $N$ new queries). In total, by combining all the queries obtained from the first and the second step, we get that there are $P \times M \times (N + 1)$ different queries, denoted as all-queries.

Third Step:
For-each query $Q_1$ in all-queries:

- For-each query $Q_2$ in all-queries:
  - if $Q_1$’s FROM clause $\neq Q_2$’s FROM clause:
    * continue.
  - else:
    * add pair $(Q_1,Q_2)$ to the dataset.

By running the third step, we obtain all the possible pairs that have the same FROM clause, and therefore, the containment rate operator is defined on them. In total, there are $P \times [M \times (N + 1)]^2$ such pairs (there are $P$ different sets of tables, and for each set there are $M \times (N + 1)$ queries from which pairs of queries are generated). By running all three steps, we obtain the dataset which includes all the query pairs generated in the third step.

After generating the dataset, we execute the queries on the IMDb database, to obtain their true result cardinalities and containment rates, while skipping query pairs that include a query with an empty result set. Using this process, we obtain the initial training set for our model. Later on, we build our evaluation test sets using the same process as will be described subsequently.
Generating training/validation sets

After generating the dataset, as described in the previous section, we obtain a file that includes all the generated pairs of queries along with their containment rates. That is, each row in the file is according to the format shown in Table 2.2.

<table>
<thead>
<tr>
<th>Query 1</th>
<th>Query 2</th>
<th>(\text{Cnt}_{\Pi}(Q1, Q2))</th>
</tr>
</thead>
</table>

Table 2.2: Dataset file structure.

From the described dataset, we generate a development set that is partitioned into two sets:

- **Training set**, used to train the model.
- **Validation set**, used to test the model convergence after each training epoch.

To create the development set, we configure our generator to randomly choose \(K\) pairs of queries from the dataset, where the chosen queries containment rates are uniformly distributed in the scale of \(\log_{10}\). That is, the development set containment rates are distributed in the following seven domains where all the domains include the same number of pairs:

\[
[1, 0.1), [0.1, 0.01), [0.01, 0.001), [0.001, 0.0001) \\
[0.0001, 0.00001), [0.00001, 0.000001), [0.000001, 0] 
\]

Note that the final domain includes the union of all the unlisted following domains, this is done since the unlisted domains only include a few pairs of queries and the majority of the dataset containment rates fall into the first six domains. This is the ideal distribution of the dataset (uniformly distributed with a logarithmic scale) since we will train our model with the q-error loss function as will be described later.

In practice, we configure our query generator with the following values:

- **\(Tables-num-threshold\) = 3** (used in the query generator’s first step).
- **\(M = 30\)** (used in the query generator’s first step).
- **\(N = 10\)** (used in the query generator’s second step).
- **\(K = 100,000\)** (development set size).
- We split the development set into 80% training set and 20% validation set.
2.2.2 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 impractical, since the order in these clauses is arbitrary. Thus, standard deep neural network architectures such as simple multi-layer perceptrons [Bro18], convolutional neural networks [Bro18], or recurrent neural networks [Bro18], 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 2.2, 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, MLPi, for each set separately. In the third stage, we estimate the containment rate Cnt% (Q1, Q2), using the representative vectors Qvec1 and Qvec2, and another specialized neural network, MLPout.

This model architecture is very practical, as it can handle any conjunctive SQL query. Using the described vector representation, all conjunctive queries are represented as one single vector (Qvec), regardless of the query’s number of tables, predicates and joins. Observe that there is only one pair of the MLP1 and MLP2 neural networks. The figure illustrates the fact that each query is represented with a set of vectors. Each vector is then presented to the same neural network separately. For further details regarding the model’s implementation, see Appendix A.4.

Figure 2.2: CRN Model Architecture.
First Stage, from \((Q_1, Q_2)\) to \((V_1, V_2)\)

Similarly to the MSCN model [KKR+19], we represent each query \(Q\) as a collection of three sets \((T, J, P)\).

- \(T\): set of the tables in \(Q\)'s FROM clause.
- \(J\): set of the join clauses in \(Q\)'s WHERE clause.
- \(P\): set of the columns predicates in \(Q\)'s WHERE clause.

Note that we ignore the query’s SELECT clause since all the queries in our model have the same SELECT * clause. In particular, even if the SELECT clause is not *, and it consists of specific columns, the cardinality is the same, i.e., the SELECT clause does not influence the cardinality of the query (as long as the DISTINCT keyword is not used).

Using sets \(T\), \(J\), and \(P\), we obtain a set of vectors \(V\) representing the query. Unlike MSCN, in our model all the vectors of set \(V\) have the same dimension and the same segmentation as depicted in Table 2.3, where:

- \(#T\): the number of tables in the whole database.
- \(#C\): the number of columns in the whole database.
- \(#O\): the number of all possible operators in predicates.

Hence, the vector size is \(#T + 4 \times #C + #O + 1\), denoted as \(L\).

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Table</th>
<th>Join</th>
<th>Column Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment</td>
<td>T-seg</td>
<td>J1-seg</td>
<td>J2-seg</td>
</tr>
<tr>
<td>Segment size</td>
<td>(#T)</td>
<td>(#C)</td>
<td>(#C)</td>
</tr>
<tr>
<td>Featurization</td>
<td>one hot</td>
<td>one hot</td>
<td>one hot</td>
</tr>
</tbody>
</table>

Table 2.3: Vector Segmentation.

Elements of sets \(T\), \(J\), and \(P\), are represented by vectors as follows (see a simple example in Section 2.2.3).
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, and all the other vector segments are zeroed.

Each join of the form \((col_1, =, col_2) \in J\) is represented as follows:

- \( col_1 \) is represented by a unique one-hot vector placed in the J1-seg segment.
- \( col_2 \) is represented by a unique one-hot vector placed in the J2-seg segment.
- All the other vector segments are zeroed.

Each predicate of the form \((col, op, val) \in P\) is represented as follows:

- \( col \) is represented by a unique one-hot vector placed in the C-seg segment.
- \( op \) is represented by a unique one-hot vector placed in the O-seg segment.
- \( val \) 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.
- All the other vector segments are zeroed.

Given input queries pair, \((Q_1, Q_2)\), we convert query \( Q_1 \) (resp., \( Q_2 \)) into sets \( T, J \) and \( P \), and each element of these sets is represented by a vector as described above, together generating set \( V_1 \) (resp., \( V_2 \)).

**Second Stage, from \((V_1, V_2)\) to \((Qvec_1, Qvec_2)\)**

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 row 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.,

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

\[
MLP_i(v) = \text{Relu}(vU_i + b_i)
\]

Where \( U_i \in \mathbb{R}^{L \times H} \), \( b_i \in \mathbb{R}^H \) are the learned weights and bias, and \( v \in \mathbb{R}^L \) 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 \( V_i \).
Third Stage, from \((Qvec_1, Qvec_2)\) to \(Cnt_{\%}(Q1, Q2)\)

Given the representative vectors of the input queries, \((Qvec_1, Qvec_2)\), we aim to predict the containment rate \(Cnt_{\%}(Q1, Q2)\) 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 \(Qvec_1\) and \(Qvec_2\). We use this function in order to provide the final network, \(MLP_{out}\), with additional information (features) 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 \([0,1]\)), \(U_{out1} \in R^{4H \times 2H}\), \(b_{out1} \in R^{2H}\) and \(U_{out2} \in R^{2H \times 1}\), \(b_{out2} \in R^1\) are the learned weights and bias, \(abs\) is the absolute 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.

**Containment Rates Featurization**

In the CRN model, the containment rates representations do not change, i.e., we do not apply any featurization to the containment rates (the output of the model) and the model is trained with the actual containment rates values.

---

\(^2\)ReLU(x) = max(0,x); see [NIGM18].

\(^3\)Sigmoid(x) = \(1/(1 + e^{-x})\); see [NIGM18].
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 [MNS09], which is the ratio between an estimated and 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.

\[
q\text{-}error(y, \hat{y}) = \begin{cases} \hat{y} - y & \text{if } \hat{y} > y \\ y - \hat{y} & \text{otherwise} \end{cases}
\]

We distinguish between three different cases:

- \( \hat{y} > y \), this means that the predicted rate was over-estimated.
- \( \hat{y} = y \), this means that the predicted rate was precise.
- \( \hat{y} < y \), this means that the predicted rate was under-estimated.

Observe that the q-error is not defined when the cardinality is zero, therefore, as described in Section 2.2.1, in creating the training and testing datasets, we skip query pairs that include a query with an empty result set.

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). For example, given true containment rate 0.0000001, and predicted rate 0.01. The MAE equals 0.0099999, which is very small, and therefore the model will largely ignore such a pair of queries. In the same way, MSE equals 0.0099999². But, on the other hand, the q-error, in this case, is very large, equals 100,000. Therefore, we decided to optimize our model using the q-error metric which yielded better results.

2.2.3 A Simple Example

In this section, we describe a simple example showing how we featurize queries in the model. That is, how we convert a query \( Q \) into a set of vectors \( V \).

Suppose that the database includes two different tables: \( A, B \), and that each table includes 3 columns: \( key, c1, c2 \). That is, in total, there are six different columns in the database. Also, suppose that there are only three optional operators: \(<, =, >\), and that all the columns values are in the range 0 to 50. We get that the vector segmentation sizes in the described database are as depicted in Table 2.4.
<table>
<thead>
<tr>
<th>Type</th>
<th>Table</th>
<th>Join</th>
<th>Column Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment</td>
<td>T-seg</td>
<td>J1-seg</td>
<td>J2-seg</td>
</tr>
<tr>
<td>Segment size</td>
<td>#T = 2</td>
<td>#C = 6</td>
<td>#C = 6</td>
</tr>
<tr>
<td></td>
<td>#C = 6</td>
<td>#O = 3</td>
<td>#C = 6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>V-seg</td>
<td></td>
</tr>
<tr>
<td>Featurization</td>
<td>one hot</td>
<td>one hot</td>
<td>one hot</td>
</tr>
<tr>
<td></td>
<td>one hot</td>
<td>one hot</td>
<td>norm.</td>
</tr>
</tbody>
</table>

Table 2.4: Vector segmentation on the example database.

**Q:** SELECT *
FROM A, B
WHERE A.key = B.key and
A.c1 < 5 and B.c1 > 5 and B.c2 = 46

Given query Q, the CRN model first calculate the Q’s sets as follows:

T = {A, B}; J = {A.key = B.key}; P = {A.c1<5, B.c1>5, B.c2=46}

As described above, each element of these sets is converted into a vector, together creating the set of vectors V representing query Q, as depicted in Table 2.5.

<table>
<thead>
<tr>
<th>Item</th>
<th>T-seg</th>
<th>J1-seg</th>
<th>J2-seg</th>
<th>C-seg</th>
<th>O-seg</th>
<th>V-seg</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10</td>
<td>000000</td>
<td>000000</td>
<td>00000</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>01</td>
<td>000000</td>
<td>000000</td>
<td>00000</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>A.key = B.key</td>
<td>00</td>
<td>100000</td>
<td>000100</td>
<td>00000</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>A.c1 &lt; 5</td>
<td>00</td>
<td>000000</td>
<td>000000</td>
<td>01000</td>
<td>100</td>
<td>0.1</td>
</tr>
<tr>
<td>B.c1 &gt; 5</td>
<td>00</td>
<td>000000</td>
<td>000000</td>
<td>000010</td>
<td>001</td>
<td>0.1</td>
</tr>
<tr>
<td>B.c2 = 46</td>
<td>00</td>
<td>000000</td>
<td>000000</td>
<td>000001</td>
<td>010</td>
<td>0.92</td>
</tr>
</tbody>
</table>

Table 2.5: Vector segmentation on the described database.

In Table 2.5 the tables’ one-hot vector entries represent tables A, B in this order. In the same way, the operators’ one-hot vector entries represent operators <, =, > in this order. The columns’ one hot vector entries represent columns A.key, A.c1, A.c2, B.key, B.c1, B.c2 in this order.

### 2.2.4 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 2.2.1. (2) Repeatedly using this training data, we train the CRN model as described in Section 2.2.2 until the mean q-error of the validation test converges to its best absolute value. In the training phase we use the early stopping technique [Pre12]. That is, we stop the training once the model achieves roughly the same mean q-error of the validation set, without any further improvements, despite additional training epochs. This is done to avoid over-fitting.
Over-fitting is the case when the model is over trained with the same training set. As a result, the model learned parameters are set too closely to fit only the training set, and they even capture the training set noise. As a result, the model fails to fit additional data or predict future unseen pairs of queries reliably.

Note that both steps are performed on an immutable snapshot of the database. To support updates and changes that may occur in the database, see Chapter 7.

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 2.2.2).

We train and test our model using the Tensor-Flow framework [ABC+16], and make use of the efficient Adam optimizer [KB15] for training the model.

2.2.5 Hyperparameter Search

To optimize the 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 2.2.2.

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 2.3). 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 2.3: The mean q-error on the validation set with different H sizes.](image-url)
2.2.6 Model Computational Costs

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

Training Time

Figure 2.4 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.

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.

Model Size

The CRN model includes all the learned parameters mentioned in Section 2.2.2 (U₁, U₂, U_{out1}, U_{out2}, b₁, b₂, b_{out1}, b_{out2}). In total, there are 2 * L * H + 8 * H² + 6 * H + 1 learned parameters, where L denotes the input vector’s dimension, and H the hidden layer size. In practice, the size of the model, when serialized to disk, is roughly 1.5MB.

![Figure 2.4: CRN model convergence of the mean q-error on the validation set.](image)
2.3 CRN Evaluation

2.3.1 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 Figure 2.5.

The Crd2Cnt Transformation

Given a cardinality estimation model\(^4\) \(M\), 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 \(Q_1\) and \(Q_2\), whose containment rate \(\text{Cnt}\%(Q_1, Q_2)\) needs to be estimated:

- Calculate the cardinality of query \(Q_1 \cap Q_2\) using \(M\).
- Calculate the cardinality of query \(Q_1\) using \(M\).
- Then, the containment rate estimate is:

\[
\text{Cnt}\%(Q_1, Q_2) = \frac{|Q_1 \cap Q_2|}{|Q_1|}
\]

Here, \(Q_1 \cap Q_2\) is the intersection query of \(Q_1\) and \(Q_2\) whose SELECT and FROM clauses are identical to \(Q_1\)’s (or \(Q_2\)’s) clauses, and whose WHERE clause is \(Q_1\)’s AND \(Q_2\)’s WHERE clauses. Operator \(|Q|\) returns the cardinality of \(Q\) with duplicates. Note that, by definition, if \(|Q_1| = 0\) then \(\text{Cnt}\%(Q_1, Q_2) = 0\).

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

We compared the CRN model predictions to those based on the other examined cardinality estimation models, using the Crd2Cnt transformation. We examined the PostgreSQL version 11 cardinality estimation component [DBM], a simple and commonly used method for cardinality estimation. In addition, we examined the MSCN model [KKR\(^+\)19]. MSCN was shown to be superior to the best methods for estimating cardinalities such as Random Sampling (RS) [BPP15, OR90] and the state-of-the-art Index-Based Join Sampling (IBJS) [LRG\(^+\)17].

\(^4\)Here "model" may refer to an ML model or simply to a method.
Comparing with MSCN

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 address this issue, we created the training dataset for the MSCN model as follows. For each pair of queries \((Q_1, Q_2)\) used in training the CRN model, we added the following two input queries to the MSCN training set:

- \(Q_1 \cap Q_2\), along with its actual cardinality.
- \(Q_1\), along with its actual cardinality.

Finally, we ensure that the training set includes only unique queries without repetition. This way, we both models, MSCN and CRN, are trained with the same information.

Comparing with PostgreSQL

Comparing with PostgreSQL does not require generating an appropriate training set, since the PostgreSQL cardinality estimation component is based on database profiling techniques and does not require training.
2.3.2 Evaluation Workloads

We evaluate CRN on the IMDb dataset as described in Section 2.2.1, using two different query workloads:

- cnt_test1, a synthetic workload generated by the same queries generator described in Section 2.2.1 (using a different random seed) with 1200 unique query pairs, with zero to two joins.

- cnt_test2, a synthetic workload generated by the same queries generator described in Section 2.2.1 (using a different random seed) with 1200 unique query pairs, with zero to five joins. This dataset is used to examine how CRN generalizes to additional joins.

<table>
<thead>
<tr>
<th>number of joins</th>
<th>0</th>
<th>1</th>
<th>3</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>cnt_test1</td>
<td>400</td>
<td>400</td>
<td>400</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1200</td>
</tr>
<tr>
<td>cnt_test2</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>1200</td>
</tr>
</tbody>
</table>

Table 2.6: Distribution of joins.

2.3.3 Evaluation Results Representations

In all the evaluation sections in this work, we use two ways to describe the evaluation results as follows:

1. Using Tables, where in each table we describe the overall test q-error results for each examined model. For this, we describe the mean, median, maximal, and different percentiles q-errors. Note that the $p$’th percentile is the q-error value below which $p\%$ of the test q-errors are found.

2. Using Box-Plots, using such plots we display the test results in readable visual way. Each plot is split according to the number of joins in the examined queries. That is, for each examined model, we show the test results while separating them according to the number of joins in the queries. Note that in all such plots described in this work, the box boundaries are at the 25th/75th percentiles and the horizontal lines mark the 5th/95th percentiles. Thus 50% of the described test results are located in the boxes, and 90% are located within the horizontal lines. Observe that we place the q-error results of the overestimated samples over q-error 1, while the q-error results of the underestimated samples under q-error 1. For more details regarding the Box-Plot anatomy, see Appendix A.1.
In all the reported results in this work, observe that the percentiles and the medians described in the tables, do not directly reflect those marked in the related Box-Plots. In the Box-Plots, the percentiles are marked according to the q-error results after being separated into two groups, the over-estimated results (above q-error 1) and under estimated results (below q-error 1). However, the percentiles, medians and averages, presented in the tables, are calculated when considering all the q-error results (under and over estimated) similarly, as positive values according to the q-error definition as described in Section 2.2.2.

2.3.4 The Quality of Estimates

Figure 2.6 depicts the q-error of the CRN model compared to the Crd2Cnt(PostgreSQL) and Crd2Cnt(MSCN) models on the cnt_test1 workload. While Crd2Cnt(PostgreSQL)'s errors are more skewed towards the positive spectrum, Crd2Cnt(MSCN) performs extremely well as does the CRN model. Observe that we make sure to train MSCN in such a way that it will predict containment rates efficiently, while the primary purpose of the MSCN model, as described in [KKR+19] is estimating cardinalities. That is, had we trained the MSCN model for its main purpose, with "independent" queries (i.e., no $Q_1 \cap Q_2$), we might have ended up with worse results for MSCN.

To provide a fuller picture, we also show the percentiles, maximum, and mean q-errors. As depicted in Table 2.7, CRN provides the best results in 75% of the tests, whereas MSCN is more robust in the margins, resulting in a better mean.

![Box-Plots of q-error comparisons](image)

Figure 2.6: Containment rate estimation q-errors on the cnt_test1 workload.
2.3.5 Generalizing to Additional Joins

In this section we examine how the CRN model generalizes to queries with a higher number of 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. Examining the results, described in Table 2.8 and Figure 2.7, the CRN model is noticeably more robust in generalizing to queries with additional joins. The mean q-error of the CRN model is smaller by a factor of almost 8 than the mean q-errors of the other models.

<table>
<thead>
<tr>
<th>Model</th>
<th>50th</th>
<th>75th</th>
<th>90th</th>
<th>95th</th>
<th>99th</th>
<th>max</th>
<th>mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crd2Cnt (PostgreSQL)</td>
<td>3.5</td>
<td>41.18</td>
<td>365</td>
<td>3399</td>
<td>268745</td>
<td>493474</td>
<td>5492</td>
</tr>
<tr>
<td>Crd2Cnt (MSCN)</td>
<td>2.84</td>
<td>7.38</td>
<td>19.95</td>
<td>41.43</td>
<td>274</td>
<td>3258</td>
<td>17.08</td>
</tr>
<tr>
<td>CRN</td>
<td>2.52</td>
<td>6.17</td>
<td>23.04</td>
<td>44.85</td>
<td>991</td>
<td>51873</td>
<td>111</td>
</tr>
</tbody>
</table>

Table 2.7: Containment rate estimation q-errors on the cnt_test1 workload.

Figure 2.7: Containment rate estimation q-errors on the cnt_test2 workload.

<table>
<thead>
<tr>
<th>Model</th>
<th>50th</th>
<th>75th</th>
<th>90th</th>
<th>95th</th>
<th>99th</th>
<th>max</th>
<th>mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crd2Cnt (PostgreSQL)</td>
<td>4.5</td>
<td>46.22</td>
<td>322</td>
<td>1330</td>
<td>39051</td>
<td>316122</td>
<td>1345</td>
</tr>
<tr>
<td>Crd2Cnt (MSCN)</td>
<td>4.1</td>
<td>17.85</td>
<td>157</td>
<td>754</td>
<td>14197</td>
<td>768051</td>
<td>1238</td>
</tr>
<tr>
<td>CRN</td>
<td>3.64</td>
<td>13.19</td>
<td>96.6</td>
<td>255</td>
<td>2779</td>
<td>56965</td>
<td>161</td>
</tr>
</tbody>
</table>

Table 2.8: Containment rate estimation q-errors on the cnt_test2 workload.
Chapter 3

Cardinality Estimation Using Containment Rates

3.1 Introduction

In this chapter 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 [LGM+15]. For further details why better cardinality estimates lead to shorter query run times, see Appendix A.3.

Three principal approaches for estimating cardinalities have emerged. (1) Using database profiling [DBM]. (2) Using histograms [AC99, BCG01]. (3) Using sampling techniques [BPP15, OR90, LRG+17]. Recently, deep learning neural networks were also used for solving this problem [KKR+19, WHT+19]. However, all these approaches, with all the many attempts to improve them, have conceptually addressed the problem directly in the same way, as a monolithic box, where the input is a query, and the output is its cardinality estimation, as described in Figure 3.1. In our proposed approach, we address the problem differently, and we obtain better estimates as described in Section 3.2.
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 [AC99, BCG01, CR94, KD98]. 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 query pool that maintains multiple previously executed queries along with their actual cardinalities, as part of the database meta information. The query 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 3.1.1).

![](Figure 3.1: Cardinality Estimation Model as a Monolithic Box.)

### 3.1.1 From Containment to Cardinality

Using a containment rate estimation model, we can obtain cardinality estimates using the Cnt2Crd transformation, as depicted in Figure 3.2.

**The Cnt2Crd Transformation**

Given a containment rate estimation model\(^1\) \(M\), we convert it to a cardinality estimation model using the Cnt2Crd transformation which returns a model \(M'\) for estimating cardinalities.

We are given a "new" query, denoted as \(Q_{\text{new}}\), as input to \(M'\) for cardinality estimation. Assume that there is an "old" query denoted as \(Q_{\text{old}}\), whose FROM clause is the same as \(Q_{\text{new}}\)'s FROM clause, that has already been executed over the database, and therefore \(|Q_{\text{old}}|\) is known. We then construct model \(M'\).

\(^1\)The term "model" may refer to an ML model or simply to a method.
The model $M'$ functions as follows:

- Calculate $x_{rate} = \text{Cnt}\%\ (Q_{old}, Q_{new})$ using $M$.
- Calculate $y_{rate} = \text{Cnt}\%\ (Q_{new}, Q_{old})$ using $M$.
- Then, the cardinality estimate equals to:

$$|Q_{new}| = \frac{x_{rate}}{y_{rate}} \times |Q_{old}|$$

provided that $y_{rate} = \text{Cnt}\%\ (Q_{new}, Q_{old}) \neq 0$.

This is true, since according to the equation used in the Crd2Cnt transformation as defined in Section 2.3.1:

$$x_{rate} = \frac{|Q_{new} \cap Q_{old}|}{|Q_{old}|}, \quad y_{rate} = \frac{|Q_{new} \cap Q_{old}|}{|Q_{new}|}$$

And therefore,

$$\frac{x_{rate}}{y_{rate}} = \frac{|Q_{new} \cap Q_{old}|}{|Q_{old}|} \times \frac{|Q_{new}|}{|Q_{new} \cap Q_{old}|} = \frac{|Q_{new}|}{|Q_{old}|}$$

where the query intersection operator, $\cap$ and operator $|Q|$, is as defined in Section 2.3.1. Given model $M$, we denote the obtained model $M'$, via the Cnt2Crd transformation, as Cnt2Crd($M$).

![Diagram](image)

Figure 3.2: Cardinality Estimation Using Containment Rate Estimation Model.
3.1.2 Query Pool

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

In addition, we may construct in advance a query 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 query pool.

Notice that we can combine both approaches (actual computing and a generator) to create the query 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 query pool contains sufficiently many queries that cover all the possible cases.

Given a query $Q$ whose cardinality is to be estimated, it is possible that we fail to find any appropriate query, in the query pool, to match with query $Q$. This happens when all the queries in the query 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 query pool includes queries with the most frequently used FROM clauses, with empty column predicates. That is, queries of the following form.

| SELECT * 
| FROM $r_1, r_2, \ldots, r_n$ 
| WHERE TRUE |

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

Consider a new query $Q_{\text{new}}$, and assume that the DBMS includes a query pool as previously described. To estimate the cardinality of $Q_{\text{new}}$ accurately, we use multiple "old" queries instead of one query, using the same Cnt2Crd transformation of Section 3.1.1, as described in Figure 3.3.

```
EstimateCardinality(Query $Q_{\text{new}}$, Query Pool $QP$):
    results = empty list
    For every pair ($Q_{\text{old}}$, $|Q_{\text{old}}|$) in $QP$:
        if $Q_{\text{old}}$'s FROM clause $\neq$ $Q_{\text{new}}$'s FROM clause:
            continue
        Calculate $x_{\text{rate}} = Cnt_{\%}(Q_{\text{old}}, Q_{\text{new}})$
        Calculate $y_{\text{rate}} = Cnt_{\%}(Q_{\text{new}}, Q_{\text{old}})$
        if $y_{\text{rate}} < \epsilon$: /* $y$ essentially zero */
            continue
        results.append($x_{\text{rate}}/y_{\text{rate}} * |Q_{\text{old}}|$)
    return $F(results)$
```

Figure 3.3: Cardinality Estimation Technique.

Algorithm EstimateCardinality considers all the matching queries whose FROM clauses are identical to $Q_{\text{new}}$'s FROM clause. For each matching query, we estimate $Q_{\text{new}}$'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. In addition, the queries in the pool can be stored in separate classes according to their FROM clauses, by using an efficient hash-based data structure.

Early Stopping

The described cardinality estimation technique considers all the matching queries to the given input query on the query 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.
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 30% outliers (trimmed removes a designated percentage of the largest and smallest values before calculating the mean).

We have examined the crd_test2 workload as described in Section 3.2.1, which includes 450 different queries including zero to five joins, and compared the cardinality estimation results, when using the Cnt2Crd(CRN) model for estimating cardinalities, with different $F$ functions. The query pool used in this experiment is as defined in Section 3.2.3.

Figure 3.4, describe the cardinality estimation results, when using Mean, 30% Trimmed Mean, and Median as the final $F$ function.

Although the improvements are relatively small, it is still possible to see that the estimates when using the trimmed mean or the median as the final $F$ function are more accurate than when using the mean. Also, in Table 3.1, it is clear that the median function obtains the most accurate results. Hence, we use the Median function as the $F$ function.

<table>
<thead>
<tr>
<th></th>
<th>50th</th>
<th>75th</th>
<th>90th</th>
<th>95th</th>
<th>99th</th>
<th>max</th>
<th>mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>3.64</td>
<td>11.46</td>
<td>51.51</td>
<td>177</td>
<td>4473</td>
<td>268671</td>
<td>753</td>
</tr>
<tr>
<td>30% Trimmed Mean</td>
<td>2.71</td>
<td>7.56</td>
<td>21.67</td>
<td>76.14</td>
<td>873</td>
<td>16065</td>
<td>64.16</td>
</tr>
<tr>
<td>Median</td>
<td>2.66</td>
<td>6.50</td>
<td>18.72</td>
<td>72.74</td>
<td>528</td>
<td>6004</td>
<td>34.42</td>
</tr>
</tbody>
</table>

Table 3.1: Estimation errors on the crd_test2 workload.

Figure 3.4: Cardinality estimation q-errors on the crd_test2 workload, while considering different Final ($F$) functions.
3.2 Cnt2Crd Evaluation

We evaluate our proposed technique for estimating cardinality, with different test sets, while using the CRN model as defined in Section 2.2.2 for estimating containment rates.

We compare our cardinality estimates with those of the PostgreSQL version 11 cardinality estimation component [DBM], and the MSCN model [KKR+19].

We train both the CRN model and the MSCN model with the same training set as described in Section 2.3.1. Also, 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 2.2.1), 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.

3.2.1 Evaluation Workloads

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

- **crd_test1**, a synthetic workload generated by the same queries generator described in Section 2.2.1 (without its final step, 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 described in Section 2.2.1 (without its final step, 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 [KKR+19]. This dataset is designed to examine how the technique generalizes to queries that were not created with the same queries’ generator used for training.

<table>
<thead>
<tr>
<th>number of joins</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>crd_test1</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>450</td>
</tr>
<tr>
<td>crd_test2</td>
<td>75</td>
<td>75</td>
<td>75</td>
<td>75</td>
<td>75</td>
<td>75</td>
<td>450</td>
</tr>
<tr>
<td>scale</td>
<td>115</td>
<td>115</td>
<td>107</td>
<td>88</td>
<td>75</td>
<td>0</td>
<td>500</td>
</tr>
</tbody>
</table>

Table 3.2: Distribution of joins.
3.2.2 Query Pool

Our technique relies on a query pool, we thus created a synthetic query pool, $QP$, generated by the same queries generator as the training data of the containment rate estimation model, as described in Section 2.2.1 (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.

<table>
<thead>
<tr>
<th>number of joins</th>
<th>0</th>
<th>1</th>
<th>3</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>$QP$</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>300</td>
</tr>
</tbody>
</table>

Table 3.3: Distribution of joins in $QP$.

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 query pool $QP$, faithfully represents a real-world situation.

3.2.3 Experimental Environment

In all the reported cardinality estimation experiments, for predicting the cardinality of a given query $Q$ in a workload $W$, we use the whole query pool $QP$ as described in Section 3.2.2 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.
3.2.4 The Quality of Estimates

Figure 3.5 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, Cnt2Crd(CRN) is competitive with MSCN that is considered a state-of-the-art model, in all the described values. As can be seen in Table 3.4, MSCN provides the best results in the margins, resulting in slightly better overall mean q-error, compared with Cnt2Crd(CRN). However, 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 3.2.5) that the Cnt2Crd(CRN) model is more robust when considering queries with more joins than in the training dataset.

<table>
<thead>
<tr>
<th></th>
<th>50th</th>
<th>75th</th>
<th>90th</th>
<th>95th</th>
<th>99th</th>
<th>max</th>
<th>mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL</td>
<td>1.74</td>
<td>3.72</td>
<td>22.46</td>
<td>149</td>
<td>1372</td>
<td>499266</td>
<td>1623</td>
</tr>
<tr>
<td>MSCN</td>
<td>2.11</td>
<td>4.13</td>
<td><strong>7.79</strong></td>
<td><strong>12.24</strong></td>
<td><strong>51.04</strong></td>
<td>184</td>
<td><strong>4.66</strong></td>
</tr>
<tr>
<td>Cnt2Crd(CRN)</td>
<td><strong>1.83</strong></td>
<td><strong>3.71</strong></td>
<td>10.01</td>
<td>18.16</td>
<td>76.54</td>
<td>1106</td>
<td><strong>9.63</strong></td>
</tr>
</tbody>
</table>

Table 3.4: Cardinality estimation q-errors on the crd\_test1 workload.
3.2.5 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 queries that have between zero and two joins.

From Tables 3.5 and 3.6, and Figure 3.6, it is clear that the Cnt2Crd(CRN) model is significantly more robust in generalizing to queries with additional joins. This is clearly illustrated in the Cnt2Crd(CRN) box plot in Figure 3.6. The boxes are almost within the same q-error interval, close to q-error 1. That is, according to the box plot anatomy, the majority of the queries were estimated with a q-error close to 1, which is the (best) q-error obtained when the estimate is 100% accurate.

In terms of overall mean q-error, the Cnt2Crd(CRN) model reduces the mean by a factor x100 compared with MSCN and by a factor of x1000 compared with PostgreSQL.

![Cardinality estimation q-errors on the crd_test2 workload.](Figure)

<table>
<thead>
<tr>
<th>Workload</th>
<th>50th</th>
<th>75th</th>
<th>90th</th>
<th>95th</th>
<th>99th</th>
<th>max</th>
<th>mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL</td>
<td>9.22</td>
<td>289</td>
<td>5189</td>
<td>21202</td>
<td>576147</td>
<td>4573136</td>
<td>35169</td>
</tr>
<tr>
<td>MSCN</td>
<td>4.49</td>
<td>119</td>
<td>3018</td>
<td>6880</td>
<td>61479</td>
<td>388328</td>
<td>3402</td>
</tr>
<tr>
<td>Cnt2Crd(CRN)</td>
<td>2.66</td>
<td>6.50</td>
<td>18.72</td>
<td>72.74</td>
<td>528</td>
<td>6004</td>
<td>34.42</td>
</tr>
</tbody>
</table>

Table 3.5: Cardinality estimation q-errors on the crd_test2 workload.
<table>
<thead>
<tr>
<th>Model</th>
<th>50th</th>
<th>75th</th>
<th>90th</th>
<th>95th</th>
<th>99th</th>
<th>max</th>
<th>mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL</td>
<td>229</td>
<td>3326</td>
<td>22249</td>
<td>166118</td>
<td>2069214</td>
<td>4573136</td>
<td>70569</td>
</tr>
<tr>
<td>MSCN</td>
<td>121</td>
<td>1810</td>
<td>6900</td>
<td>25884</td>
<td>83809</td>
<td>388328</td>
<td>6801</td>
</tr>
<tr>
<td>Cnt2Crd(CRN)</td>
<td><strong>4.28</strong></td>
<td><strong>10.84</strong></td>
<td><strong>43.71</strong></td>
<td><strong>93.11</strong></td>
<td><strong>1103</strong></td>
<td><strong>6004</strong></td>
<td><strong>61.26</strong></td>
</tr>
</tbody>
</table>

Table 3.6: Cardinality estimation q-errors on the crd.test2 workload considering only queries with three to five joins.

To highlight these improvements, we describe, in Table 3.7 and Figure 3.7, the mean and median q-error for each possible number of joins separately (note the logarithmic y-axis scale in Figure 3.7).

The known cardinality estimation models suffer from producing under-estimated results and errors that grow exponentially as the number of joins increases [CK91]. 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).

**Why Cnt2Crd(CRN) is more robust**

The reason why the Cnt2Crd(CRN) model successfully generalizes to additional joins lies in its use of the query pool. The query 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. In addition, a key point why our proposed technique manages to estimate the cardinalities more robustly hides in the following equation, which is used in the Cnt2Crd transformation (described in Section 3.1.1):

\[
|Q_{new}| = \frac{Cnt_{\%}(Q_{old}, Q_{new})}{Cnt_{\%}(Q_{new}, Q_{old})} \times |Q_{old}|
\]

Both containment rates \( Cnt_{\%}(Q_{old}, Q_{new}) \) and \( Cnt_{\%}(Q_{new}, Q_{old}) \), are estimated using the CRN model. Even though the CRN model can produce wrong estimates, it seems that the estimates for both containment rates are wrong by a "similar magnitude" in terms of q-error. Thus these errors are "cancelled out" in the fraction, resulting in accurate cardinality estimates.
3.2.6 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 [KKR+19]. As shown in Table 3.8, Cnt2Crd(CRN) is clearly more robust than MSCN and PostgreSQL in all the described percentiles. Examining Figure 3.8, it is clear that the Cnt2Crd(CRN) model is significantly more robust with queries with 3 and 4 joins. Recall that the QP query 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.
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 [KKR+19]. 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 3.8, 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 reasons described in Section 3.2.5.

### Table 3.8: Cardinality estimation q-errors on the scale workload.

<table>
<thead>
<tr>
<th></th>
<th>50th</th>
<th>75th</th>
<th>90th</th>
<th>95th</th>
<th>99th</th>
<th>max</th>
<th>mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL</td>
<td>2.62</td>
<td>15.42</td>
<td>183</td>
<td>551</td>
<td>2069</td>
<td>233863</td>
<td>586</td>
</tr>
<tr>
<td>MSCN</td>
<td>3.76</td>
<td>16.84</td>
<td>100</td>
<td>448</td>
<td>3467</td>
<td>47847</td>
<td>204</td>
</tr>
<tr>
<td>Cnt2Crd(CRN)</td>
<td>2.53</td>
<td>5.88</td>
<td>24.02</td>
<td>95.26</td>
<td>598</td>
<td>19632</td>
<td>69.85</td>
</tr>
</tbody>
</table>

Figure 3.8: Cardinality estimation q-errors on the scale workload.
3.3 Improving Existing Cardinality Estimation Models

In this section we describe how existing cardinality estimation models can be improved using the idea underlying 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 3.1.3.

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 2.3.1.

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.

3.3.1 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 2.3.1. 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 3.1.3, which uses a query 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 query pool, as depicted in Figure 3.9.

For clarity, given cardinality estimation model $M$, we denote the model $M''$ described above, i.e., model Cnt2Crd(Crd2Cnt($M$)), as Improved $M$ model.
Figure 3.9: A novel approach, from cardinality estimation to containment rate estimation, and back to cardinality estimation by using a query pool.
3.3.2 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 3.2.1, as it includes the most number of joins. Tables 3.9 and 3.10 depict the estimates when using directly the PostgreSQL and MSCN models, respectively, 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 query pool, when these models are converted back to estimate cardinalities, they obtain better estimates.

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 creating a query pool in the database, cardinality estimates can be improved significantly.

![Figure 3.10: Cardinality estimation q-errors on the crd_test2 workload. Comparing the basic and the Improved PostgreSQL model.](image)

<table>
<thead>
<tr>
<th></th>
<th>50th</th>
<th>75th</th>
<th>90th</th>
<th>95th</th>
<th>99th</th>
<th>max</th>
<th>mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL</td>
<td>9.22</td>
<td>289</td>
<td>5189</td>
<td>21202</td>
<td>576147</td>
<td>4573136</td>
<td>35169</td>
</tr>
<tr>
<td>Improved PostgreSQL</td>
<td>2.61</td>
<td>19.3</td>
<td>155</td>
<td>538</td>
<td>17697</td>
<td>1892732</td>
<td>5081</td>
</tr>
</tbody>
</table>

Table 3.9: Cardinality estimation q-errors on the crd_test2 workload. Comparing the basic and the Improved PostgreSQL model.
Figure 3.11: Cardinality estimation q-errors on the crd_test2 workload.
Comparing the basic and the Improved MSCN model.

<table>
<thead>
<tr>
<th></th>
<th>50th</th>
<th>75th</th>
<th>90th</th>
<th>95th</th>
<th>99th</th>
<th>max</th>
<th>mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSCN</td>
<td>4.49</td>
<td>119</td>
<td>3018</td>
<td>6880</td>
<td>388328</td>
<td>3402</td>
<td></td>
</tr>
<tr>
<td>Improved MSCN</td>
<td>2.89</td>
<td>7.43</td>
<td>25.26</td>
<td>55.73</td>
<td>196</td>
<td>3184</td>
<td>27.78</td>
</tr>
</tbody>
</table>

Table 3.10: Cardinality estimation q-errors on the crd_test2 workload.
Comparing the basic and the Improved MSCN model.

### 3.3.3 Improved Models vs. Cnt2Crd(CRN)

Using the crd_test2 workload, we examine how our technique improves PostgreSQL and MSCN, compared with Cnt2Crd(CRN). Examining Table 3.11, 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.

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

Table 3.11: Cardinality estimation q-errors on the crd_test2 workload.
Comparing the Improved models and Cnt2Crd(CRN).
Figure 3.12: Cardinality estimation q-errors on the crd_test2 workload. Comparing with all the examined models.
3.4 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 query pool, and calculating the final function $F$ on these results to obtain the predicted cardinality, as described in Section 3.1.3. Therefore, the larger the query pool is, the longer the prediction time is, and the more accurate the predictions are.

Table 3.12 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 query pool ($QP$) equally distributed over all the possible FROM clauses in the database, while using the same final function $F$ (the Median function).

The results described in Table 3.12 show that having bigger query pools is not always necessary. For example, the results when the pool is of size 150 are almost the same as when using bigger query pool. Additionally, using bigger query pool does not always lead to better cardinality estimates, as can be seen in the results when using query pool of size 600. That is, after a certain point, the estimates stop improving and reach their best.

Observe that when using smaller query pools, the cardinality prediction time is shortened accordingly. Therefore, we conclude that in order to obtain good estimates while keeping the prediction time practical, using moderate sizes of query pool is appropriate. Alternatively, we may adapt the early stopping technique described in Section 3.1.3. In this case, the query pool size will not have a direct impact on the prediction time, as we stop the prediction process once we cross the defined threshold.

<table>
<thead>
<tr>
<th>QP Size</th>
<th>50</th>
<th>100</th>
<th>150</th>
<th>200</th>
<th>250</th>
<th>300</th>
<th>600</th>
</tr>
</thead>
<tbody>
<tr>
<td>Median q – error</td>
<td>3.68</td>
<td>2.55</td>
<td>2.63</td>
<td>2.55</td>
<td>2.61</td>
<td>2.66</td>
<td>2.57</td>
</tr>
<tr>
<td>Mean q – error</td>
<td>1894</td>
<td>90</td>
<td>41</td>
<td>40</td>
<td>35</td>
<td>34</td>
<td>35</td>
</tr>
<tr>
<td>Prediction Time</td>
<td>3.2ms</td>
<td>7.1ms</td>
<td>9.8ms</td>
<td>11.3ms</td>
<td>14.5ms</td>
<td>16.1ms</td>
<td>28.3ms</td>
</tr>
</tbody>
</table>

Table 3.12: Median and mean estimation q-errors on the crd_test2 workload, and the average prediction time, considering different query pool (QP) sizes.
In table 3.13, we compare the average prediction time for estimating the cardinality of a single query using all the examined models (when using the whole QP query pool of size 300). The default MSCN model is the fastest model, since it directly estimates the cardinalities without using a query pool. The Cnt2Crd(CRN) model is the fastest among all the models that use a query 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 2.3.1, whereas the CRN model directly obtains a containment rate for the two queries in one pass within 0.5ms (as described in Section 2.2.6).

Although the prediction time of the models that use a query 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 3.13, we used a query pool (QP) of size 300. We could have used a smaller pool (or adapt the early stopping technique as mentioned in Section 3.1.3), resulting in faster prediction time, and still obtain better results, as depicted in Table 3.12. Furthermore, all the the models that use a query pools may be easily parallelized as discussed in Section 3.1.3, thus reducing the prediction time (in the reported tests we ran these models serially).

<table>
<thead>
<tr>
<th>Model</th>
<th>Prediction Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL</td>
<td>1.75ms</td>
</tr>
<tr>
<td>MSCN</td>
<td>0.5ms</td>
</tr>
<tr>
<td>MSCN with 1000 samples</td>
<td>33ms</td>
</tr>
<tr>
<td>Improved PostgreSQL</td>
<td>70ms</td>
</tr>
<tr>
<td>Improved MSCN</td>
<td>35ms</td>
</tr>
<tr>
<td>Cnt2Crd(CRN)</td>
<td>16ms</td>
</tr>
</tbody>
</table>

Table 3.13: Average prediction time of a single query.
Chapter 4

Estimating Set-Theoretic Cardinality

4.1 Introduction

Despite the superiority of Cnt2Crd(CRN) for estimating cardinalities using containment rates, our technique is designed to estimate cardinalities with duplicates only. Thus, ignoring the importance of the set-theoretic cardinality estimation in query optimization.

This lack of support for estimating the theoretic cardinalities, is not only present in Cnt2Crd(CRN), it also present in several cardinality estimation models, e.g., MSCN. As a result, the applicability of such models in the context of a real world systems, is questionable.

We would like to extend such models, without changing them, to produce set-theoretic cardinality estimates. Therefore, in this chapter, we propose a technique to extend such models, without changing them, to produce set-theoretic cardinality estimates. To this end, we design a NN based model, PUNQ, which is tailored to estimate the uniqueness rate of its input query (see Section 4.2). Using the PUNQ model, we can transform any limited model, e.g., the MSCN model [KKR+19], to estimate cardinalities without duplicates (thereby, for example, supporting queries with the DISTINCT keyword).
4.2 Uniqueness Rate Definition

We define the uniqueness rate of query $Q$ on a specific database $D$, as follows.

The uniqueness rate of query $Q$ equals to $x\%$ on database $D$ if precisely $x\%$ of $Q$’s execution result rows (with duplicates) on database $D$ are unique. The uniqueness rate is formally a function from $Q \times D$ to $[0,1]$, where $Q$ is the set of all queries, and $D$ of all databases. This function can be directly calculated using the cardinality of the result of query $Q$ as follows:

$$x\% = \frac{||Q(D)||}{|Q(D)|}$$

where, $Q(D)$ denotes $Q$’s execution result (with duplicates) on database $D$. The operator $|| \cdot ||$ returns the set theoretic cardinality, and the operator $| \cdot |$ returns the cardinality with duplicates. (In case $Q$’s execution result is empty, then $Q$ has 0% unique rows). For clarity, we do not mention the specific database $D$, that it is usually clear from context.

Example 4.2.1. Consider database $D$, and the following SQL queries:

<table>
<thead>
<tr>
<th>From</th>
<th>Destination</th>
<th>duration</th>
<th>stops</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLV</td>
<td>YYZ</td>
<td>12:30</td>
<td>1</td>
</tr>
<tr>
<td>JFK</td>
<td>TLV</td>
<td>10:30</td>
<td>0</td>
</tr>
<tr>
<td>YYZ</td>
<td>TXL</td>
<td>08:30</td>
<td>2</td>
</tr>
<tr>
<td>JFK</td>
<td>TLV</td>
<td>10:00</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.1: Database D.

Q1: SELECT from, destination FROM D WHERE stops <= 1
Q2: SELECT * FROM D WHERE destination == TXL AND stops == 2

Query Q1 execution result on database $D$ is the highlighted portion of D. By definition, query Q1’s cardinality with duplicates equals to 3, and the cardinality without duplicates equals to 2 (note that the last row is duplicated in the result, while the other rows are unique). Thus, Q1’s uniqueness rate equals to 66.67%. This is true, since by definition:

$$Q1’s\ Uniqueness\ Rate = \frac{||Q1(D)||}{|Q1(D)|} = \frac{2}{3} = 0.6667$$

Query Q2 execution result includes one row only (the row with stops = 2). Hence, Q2’s uniqueness rate equals to 100%. That is, all the query’s results are unique.
Suppose we are given a cardinality estimate $C$ with duplicates of some query $Q$ on $D$, obtained from some cardinality estimation model. By uniqueness rate definition, we can easily obtain the set-theoretic cardinality estimation simply by multiplying the cardinality estimate $C$ with the uniqueness rate of query $Q$. Therefore, using the uniqueness rates of queries we can simply transform any limited cardinality estimation method to one that estimates set-theoretic cardinality. Hence, in the following sections we aim to estimate accurately and efficiently the uniqueness rates of queries. For this, we use the subsequently described PUNQ model.

### 4.3 Learned Uniqueness Rates

In applying machine learning to the uniqueness-rate estimation problem, we face the following common issues that are faced by any other deep learning technique, similarly to those that we face with the CRN model, as described in Section 2.2.

- Cold start problem: obtaining the initial training dataset.
- Model: determining which supervised algorithm/model should be used.
- Featurization: determining useful representations of the inputs and the outputs of the model.

In the following sections we describe how each issue is addressed.

![Figure 4.1: Uniqueness-rate Estimation Model as a Monolithic Box.](image-url)
4.3.1 Cold Start Problem

Defining the Database

We generated a training-set of queries, and later on evaluated our model on it, using the IMDb database. This database is challenging for cardinality estimation. It was used in evaluating the methods we extend; our own model, CRN, and the MSCN model.

Generating the Development Dataset

We obtain an initial training corpus using a specialized queries generator. The queries generator is constructed similarly to the one used for training the CRN model as described in Section 2.2.1. The queries generator forms the query’s WHERE and FROM clauses, identically to the query’s generator first and second steps as described in Section 2.2.1. Note that the query’s generator last step, that of choosing pairs of SQL queries, is irrelevant for this task.

The queries generator that is described in Section 2.2.1 generates queries with a SELECT * clause. However, the specifics of the SELECT clause are extremely important for the task of predicting uniqueness rates. Therefore, we reconfigure the queries generator with the following additional step. After generating the query with the SELECT * clause, we replace the * in the SELECT clause by randomly choosing the number of columns $c$ to be in the query’s SELECT clause, and then choosing at random $c$ different columns from the possible columns of the chosen tables.

In order to avoid a combinatorial explosion of the number of possible queries, and to facilitate the learning of the model, we create queries with up to two joins only. We also examine how the learned models, learned over queries with this small number of joins, generalize to a larger number of joins.

Once generating the dataset of queries, we execute the dataset queries on the IMDb database. This allows us to obtain their true uniqueness rates (ground truth). In this process, we obtain an initial training set for our model. It consists of 20,000 queries with zero to two joins. We split the training samples into 80% training samples and 20% validation samples.
4.3.2 Model

We form the PUNQ model for estimating the uniqueness rates (the percentage of rows in the input query’s result, with duplicates, that are unique). PUNQ was inspired by the architecture of the CRN model (Section 2.2). Similarly to CRN, the PUNQ model has three main stages. In the first stage, it converts the input query into a set \( V \) of vectors. In stage two, using a specialized neural network, \( MLP_{mid} \), the set \( V \) of vectors is transformed into \( Qvec \), a single representative vector that represents the whole input query. In stage three, the percentage rate of unique rows is estimated using \( MLP_{out} \), a two-layers neural network.

Observe that there is a single \( MLP_{mid} \) neural network. The figure illustrates the fact that each query is represented with a set of vectors. Each vector is then presented to the same neural network separately. For further details regarding the model’s implementation, see Appendix A.4.

![Figure 4.2: PUNQ Model Architecture.](image)

First Stage, from \( Q \) to \( V \)

We represent a query \( Q \) as a collection of four sets \((A, T, J, P)\), similarly to CRN model, except of that we have an additional set, set \( A \).

- \( A \): set of the attributes in \( Q \)’s SELECT clause.
- \( T \): set of the tables in \( Q \)’s FROM clause.
- \( J \): set of the join clauses in \( Q \)’s WHERE clause.
- \( P \): set of the columns predicates in \( Q \)’s WHERE clause.

Each element in each of the 4 sets is then represented by a vector. Together these vectors form the set \( V \) of vectors. To facilitate learning, all the vectors in \( V \) are of the same dimension and the same format as depicted in Table 4.2, where; \#\( T \) is the number of tables in the whole database. \#\( C \) is the number of columns in the whole database. \#\( O \) is the number of all possible operators in predicates.
Similarly to CRN model, for each element of the $A, T, P$, and $J$ sets, we create a vector of the same format as described in Table 4.2, that includes zeros in all its segments, except for those that are relevant for representing its element, as follows.

Elements of set $A$, are represented in the A-seg segment. Elements of set $T$, are represented in the T-seg segment. Elements $(col1, =, col2)$ of set $J$ are represented in two segments. $col1$ and $col2$ are represented in J1-seg and J2-seg segments, respectively. Elements $(col, op, val)$ of set $P$ are represented in three segments. $col$ and $op$ are represented in the C-seg and O-seg segments, respectively; $val$ is represented in the V-seg segment.

All the segments, except for the V-seg, are represented by a binary sequence with a single non-zero entry, uniquely identifying the corresponding element (one-hot vector). For example, the A-seg segment, is represented with one-hot vector uniquely identifying the corresponding attribute name.

In the V-seg segment, we represent the predicates’ values similarly to CRN. That is, these values are represented using their normalized value (a value in $[0,1]$).

### Second Stage, from $V$ to $Qvec$

Given the set of vectors $V$, we present each vector of $V$ as input to $MLP_{mid}$, a fully-connected one-layer neural network. $MLP_{mid}$ transforms each presented vector into a vector of dimension $H$. The final representation for the whole set $V$ is then given by $Qvec$, the average over the individual transformed representations of its elements, i.e.,

$$Qvec = \frac{1}{|V|} \sum_{v \in V} MLP_{mid}(v)$$

$$MLP_{mid}(v) = ReLU(vU_{mid} + b_{mid})$$

where $U_{mid} \in R^{L \times H}$, $b_{mid} \in R^H$ are the learned weights and bias, and $v \in R^L$ is the input row vector. We choose an average (instead of, say, sum) to facilitate generalization to a different number of elements in $V$. Had we used sum, the overall magnitude of $Qvec$ would depend on the cardinality of the set $V$. 52
Third Stage, from $Qvec$ to Uniqueness rate

Given the representative vector of the input query, $Qvec$, we aim to predict the uniqueness rate. To do so, we use $MLP_{out}$, a fully-connected two-layer neural network, to compute the estimated uniqueness rate of the input query. $MLP_{out}$ takes as input the $Qvec$ vector of size $H$. The first layer in $MLP_{out}$ converts the input vector into a vector of size $0.5H$. The second layer converts the obtained vector of size $0.5H$, into a single value representing the uniqueness rate. This architecture allows $MLP_{out}$ to learn the uniqueness rate function.

$$\hat{y} = MLP_{out}(Qvec)$$

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

where $\hat{y}$ (a value in $[0,1]$) is the estimated uniqueness rate, $U_{out1} \in \mathbb{R}^{H \times 0.5H}$, $b_{out1} \in \mathbb{R}^{0.5H}$ and $U_{out2} \in \mathbb{R}^{0.5H \times 1}$, $b_{out2} \in \mathbb{R}^{1}$ are the learned weights and bias.

We use the empirically effective and fast to evaluate $ReLU^{1}$ activation function for hidden layers units in all our neural networks. The uniqueness rate values are in the range $[0,1]$. Therefore, in the final step, we apply the $Sigmoid^{2}$ activation function in the second layer to output a float value in this range.

Uniqueness Rates Featurization

In the PUNQ model, we do not apply any featurization to the uniqueness rates (the output of the model) and the model is trained with the actual uniqueness rate values without any intermediate featurization steps.

4.3.3 Training and Testing Interface

The PUNQ model building includes two main steps, performed on an immutable snapshot of the database, similarly to the CRN model as described in Section 2.2.4. (1) Generating a random training set. (2) Training the PUNQ model. Note that we use the same loss function, q-error, that was used for evaluating the CRN model. Following the training phase, and in order to predict the uniqueness rate of an input query, the (encoded) query is simply presented to the PUNQ model, and the model outputs the estimated uniqueness rate, as described in Section 4.3.2. We train and test our model using the Tensor-Flow framework [ABC+16], using the Adam training optimizer [KB15].

---

1$ReLU(x) = \max(0,x)$; see [NIGM18].
2$Sigmoid(x) = 1/(1 + e^{-x})$; see [NIGM18].
4.3.4 Hyperparameter Search and Model Costs

To optimize the model’s performance, we searched the hyperparameter space. We considered different settings, where we varied the number of the batch size, hidden layer size, and learning rate.

Similarly to the CRN model, it turned out that the hidden layer size has the most impact on the model’s accuracy for the validation test. Until it reaches the best result, the bigger the hidden layer size is, the more accurate the model is on the validation test. Afterwards, quality declines due to over-fitting. Further, both the learning rate and the batch size mainly influence the training convergence behavior rather than the model accuracy. Averaged over 5 runs over the validation set, the best configuration has a 128 batch size, a 512 hidden layer size, and a 0.001 learning rate. These settings are thus used throughout our model evaluation.

4.3.5 Model Computational Costs

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

Training Time

The PUNQ model requires almost 200 passes on the training set to converge to a mean q-error of around 3.5. On average, measured across six runs, a training run with 200 epochs takes nearly 30 minutes.

Prediction Time

The prediction process is dominated by converting the input queries into the corresponding vectors, and presenting these vectors as input to the PUNQ model. On average, the prediction time is 0.05ms per query, including the overhead introduced by the Tensor-Flow framework.

Model Size

The PUNQ model includes all the learned parameters mentioned in Section 4.3.2 \((U_{mid}, U_{out1}, U_{out2}, b_{mid}, b_{out1}, b_{out2})\). In total, there are \(L \times H + 0.5 \times H^2 + H + 1\) learned parameters, where \(L\) denotes the input vector’s dimension, and \(H\) the hidden layer size. In practice, the size of the model, when serialized to disk, is roughly 0.2MB.
4.4 PUNQ Evaluation

In this section, we examine how well our extension model (PUNQ model) transforms limited cardinality estimation models that predict cardinalities with duplicates into ones that predict set-theoretic cardinalities. To this end, we extended four different limited models, MSCN, CRN (in this Chapter referring to the Cnt2Crd(CRN) model), Improved PostgreSQL, and Improved MSCN. Each model was extended as detailed below. For clarity, we denote the extended model of model $M$ as $\text{PUNQ}(M)$. Given query $Q$ whose set-theoretic cardinality needs to be estimated (i.e., estimation without duplicates), $\text{PUNQ}(M)$ functions as follows.

- calculate $C = \text{model's } M$ estimated cardinality with duplicates of query $Q$.
- calculate $U = \text{PUNQ uniqueness rate estimation of query } Q$.
- return $U \cdot C$, the predicted set-theoretic cardinality.

We evaluate each of the four abovementioned limited models over two test workloads. Each model was extended to estimate cardinality without duplicates, using the PUNQ model. In addition, we compare the results with those of the PostgreSQL version 11 cardinality estimation component [DBM], as PostgreSQL supports all SQL queries. To estimate cardinality without duplicates using PostgreSQL, we added the DISTINCT keyword in the queries’ select clauses, and then used the ANALYZE command to obtain the estimated (set-theoretic) cardinalities.

In order to ensure a fair comparison between the models, we trained the CRN, MSCN and the PUNQ models with the same types of queries, where all the models training sets were generated using similar queries generators. In particular, these models were trained with conjunctive queries with only up to two joins.

4.4.1 Evaluation Workloads

The evaluation uses the IMDb dataset, over two query workloads, UnqCrd_test1 and UnqCrd_test2. Each workload includes 450 queries that were generated by the same queries generator described in Section 4.3.1 (using a different random seed). Both workloads are equally distributed in the number of joins.

<table>
<thead>
<tr>
<th>number of joins</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>UnqCrd_test1</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>450</td>
</tr>
<tr>
<td>UnqCrd_test2</td>
<td>75</td>
<td>75</td>
<td>75</td>
<td>75</td>
<td>75</td>
<td>75</td>
<td>450</td>
</tr>
</tbody>
</table>

Table 4.3: Distribution of joins.
4.4.2 The Quality of Estimates

We examined the UnqCrd_test1 workload on two limited models, MSCN and CRN. Recall that this workload includes queries with up to two joins, and that the MSCN, CRN and PUNQ models were trained over such conjunctive queries.

Figure 4.3 and Table 4.4 depict the PUNQ model uniqueness rates estimations. It is clear that the PUNQ model estimates are very accurate, where 75% of the examined queries uniqueness rates were predicted accurately with a q-error smaller than 1.93. That is, the ratio between the estimated rates and the actual rates does not exceed 1.93, whether it is over or under estimated. These results are similar for queries with 0, 1, and 2 joins.

The fact that the PUNQ model uniqueness rate estimates ($U$) are very accurate, along with the fact that the CRN and the MSCN models cardinality estimates with duplicates ($C$) are accurate as well, is thus reflected in accurate cardinality estimates without duplicates, namely ($U \cdot C$).

![Figure 4.3: Uniqueness rates estimation q-errors on the UnqCrd_test1 workload.](image)

<table>
<thead>
<tr>
<th></th>
<th>50th</th>
<th>75th</th>
<th>90th</th>
<th>95th</th>
<th>99th</th>
<th>max</th>
<th>mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUNQ</td>
<td>1.12</td>
<td>1.93</td>
<td>3.71</td>
<td>5.59</td>
<td>13.65</td>
<td>139</td>
<td>2.27</td>
</tr>
</tbody>
</table>

Table 4.4: Uniqueness rates estimation q-errors on the UnqCrd_test1 workload.
Examining the results in Figure 4.4 and Table 4.5, it is apparent that when the CRN and MSCN models are extended to estimate cardinalities without duplicates using the PUNQ model, the estimates are still very accurate. Observe that the models we extend were originally tailored to estimate cardinalities with duplicates only. Note that the PostgreSQL model was not extended using the PUNQ model, as it supports any SQL queries as is. Interestingly, it obtained the worst results among the examined models.

Additionally, it is clear that the PUNQ(CRN) model is more robust than PUNQ(MSCN) in 99% of the test. This is apparently the case since the original limited model, CRN, was more robust in estimating cardinalities with duplicates, compared with MSCN, as shown in Chapter 3.

![Figure 4.4: Cardinality estimation q-errors on the UnqCrd_test1 workload.](image)

<table>
<thead>
<tr>
<th></th>
<th>50th</th>
<th>75th</th>
<th>90th</th>
<th>95th</th>
<th>99th</th>
<th>max</th>
<th>mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL</td>
<td>1.82</td>
<td>4.93</td>
<td>27.26</td>
<td>163</td>
<td>926</td>
<td>372207</td>
<td>1214</td>
</tr>
<tr>
<td>PUNQ(MSCN)</td>
<td>2.13</td>
<td>4.83</td>
<td>10.05</td>
<td>16.37</td>
<td>60.53</td>
<td>222</td>
<td>5.66</td>
</tr>
<tr>
<td>PUNQ(CRN)</td>
<td>2.05</td>
<td>4.27</td>
<td>9.98</td>
<td>15.24</td>
<td>49.11</td>
<td>1037</td>
<td>7.49</td>
</tr>
</tbody>
</table>

Table 4.5: Cardinality estimation q-errors on the UnqCrd_test1 workload.
4.4.3 Generalizing to Additional Joins

In this section we examine how the PUNQ model generalizes to queries with more than two joins, and how limited models adapt to such queries under the PUNQ model extension. To do so, we use the UnqCrd_test2 workload which includes queries with zero to five joins. Recall that we trained the MSCN, CRN, and PUNQ models with conjunctive queries that have only up to two joins. To examine the limited cardinality estimation models more broadly, we used another limited models that we introduced in Chapter 3, the Improved PostgreSQL model, and the Improved MSCN model, in addition to the CRN and MSCN models.

Figure 4.5 and Table 4.6 depict the PUNQ model uniqueness rate estimations. Note that the PUNQ model estimates are very accurate when considering queries with zero to two joins, whereas the estimates tend to over-estimate as the number of joins increases, starting from 3 joins. This is to be expected as the PUNQ model was trained with queries with up to two joins only. Nevertheless, the overall estimates are still relatively accurate.

![Figure 4.5: Uniqueness rates estimation q-errors on the UnqCrd_test2 workload.](image)

<table>
<thead>
<tr>
<th></th>
<th>50th</th>
<th>75th</th>
<th>90th</th>
<th>95th</th>
<th>99th</th>
<th>max</th>
<th>mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUNQ</td>
<td>1.75</td>
<td>3.55</td>
<td>9.9</td>
<td>18.08</td>
<td>109</td>
<td>214</td>
<td>6.62</td>
</tr>
</tbody>
</table>

Table 4.6: Uniqueness rates estimation q-errors on the UnqCrd_test2 workload.
Figure 4.6: Cardinality estimation q-errors on the UnqCrd_test2 workload.

Figure 4.6 provides a fuller picture of how the limited models estimates are affected when they are extended to estimate cardinalities without duplicates. In the upper part of Figure 4.6, we present the limited models’ cardinality estimates errors (with duplicates), i.e., the models’ original estimates. In the bottom part, we show the models’ estimates errors on the same queries, when the models are extended to estimate set-theoretic cardinalities using the PUNQ model.

According to box-plots definition, 50% of the test queries are located within the box boundaries, and 90% are located between the horizontal lines. Observe that the boxes’ boundaries and the horizontal lines were hardly changed in the extended models box-plots, compared with the original models box-plots. We conclude that for each model and for each number of joins, the majority of the queries are located in roughly the same error domains. That is, the models’ accuracies were hardly changed after extending them using the PUNQ model. They maintained almost the same quality of estimates for set-theoretic cardinality, as the original models’ quality for estimating cardinality with duplicates.

Significant changes in the models quality of estimates may be observed in queries with more than two joins. In such queries, the PUNQ model suffers from slightly overestimated uniqueness rates, which directly affect the extended models set-theoretic cardinality estimates. This is depicted via their box-plots horizontal lines moving a bit upwards as compared with the box-plots of the original models estimates.

Table 4.7 shows the cardinality estimates errors percentiles and means when using the examined extended models.

---

3Here there are two potential sources of errors, the estimate with duplicates and PUNQ.
<table>
<thead>
<tr>
<th>Model</th>
<th>50th</th>
<th>75th</th>
<th>90th</th>
<th>95th</th>
<th>99th</th>
<th>max</th>
<th>mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL</td>
<td>3.95</td>
<td>34.95</td>
<td>651</td>
<td>2993</td>
<td>20695</td>
<td>331152</td>
<td>2029</td>
</tr>
<tr>
<td>PUNQ(MSCN)</td>
<td>4.85</td>
<td>59.65</td>
<td>1189</td>
<td>5476</td>
<td>86046</td>
<td>288045</td>
<td>3430</td>
</tr>
<tr>
<td>PUNQ(Improved Post.)</td>
<td>3.51</td>
<td>20.97</td>
<td>134</td>
<td>444</td>
<td>9770</td>
<td>136721</td>
<td>563</td>
</tr>
<tr>
<td>PUNQ(Improved MSCN)</td>
<td>3.31</td>
<td>10.81</td>
<td>39.95</td>
<td>90.98</td>
<td>778</td>
<td>3069</td>
<td>35.71</td>
</tr>
<tr>
<td>PUNQ(CRN)</td>
<td>3.15</td>
<td>10.35</td>
<td>33.73</td>
<td>88.61</td>
<td>224</td>
<td>3298</td>
<td>24.46</td>
</tr>
</tbody>
</table>

Table 4.7: Cardinality estimation q-errors on the UnqCrd_test2 workload.

Examining Table 4.7 and Figure 3.12, it is clear that the PUNQ(CRN) and PUNQ(Improved MSCN) models are significantly more robust in generalizing to queries with additional joins as compared with the other examined models. In particular, the two models are more robust than PostgreSQL. This is despite the fact that PostgreSQL can estimate cardinalities without duplicates by default, without the need to extend it using the PUNQ model, as was done for the other limited models.

These results highlight the usefulness of the PUNQ model in extending limited models. PUNQ therefore presents a simple tool to extend limited models, while keeping their quality of set-theoretic estimates roughly at the same quality.

### 4.5 Cardinality Prediction Time

To estimate cardinalities without duplicates by extending limited cardinality estimation models, we used the PUNQ model along with the estimated cardinalities of these models. Therefore, the prediction time of cardinalities without duplicates, by extending limited models, consists of two parts. First, the prediction time of the limited model. Second, the prediction time of the uniqueness rate, obtained from the PUNQ model.

The prediction times of the limited models vary, depending on the model itself (The average prediction time for each such model is depicted in Table 3.13 in Section 3.4). As stated in Section 4.3.5, the average uniqueness rate prediction time using the PUNQ model takes 0.05ms, per query. Therefore, the prediction time overhead due to using the PUNQ model, when extending limited models, is minor (0.05ms), as compared with the limited models’ prediction times (of cardinalities with duplicates). Also, estimating both the cardinality with duplicates and the uniqueness rate can be done in parallel. In this case, the PUNQ model overhead (0.05ms) will be eliminated, as all the limited models require more than 0.05ms to obtain their estimates (with duplicates).
Chapter 5

Supporting AND, OR and NOT operators

5.1 Introduction

The cardinality estimation techniques presented thus far are designed to estimate cardinalities of conjunctive queries only (i.e., queries that only use the AND operator). This limits the supported queries, as the frequently used queries that use the AND, OR and NOT operators, are not supported. Such queries are denoted in this work as general queries.

This lack of support for general queries is not only present in our proposed techniques, it is also present in several cardinality estimation models, e.g., MSCN. As a result, the applicability of such models in a real world setting is questionable.

We would like to extend such models, without changing them, to support general queries. Therefore, in this chapter we introduce a uniform alternative approach for estimating cardinalities for general queries, namely the recursive GenCrd algorithm. Using this approach, we can use any limited model, that only supports conjunctive queries (i.e., AND), to estimate cardinalities for queries with any combination of the AND, OR, and NOT operators.

The end result of introducing the GenCrd algorithm is efficiently and effectively extending estimation capabilities to a vaster class of queries. This eliminates the need to train models from scratch on this gigantic class of queries. It also factors out this code for any optimizer which results in a better software architecture.
5.2 The GenCrd Algorithm

Consider estimating the cardinality of a general query $Q$. The GenCrd algorithm relies mainly on two observations.

5.2.1 First Observation

Given a general query $Q$ that includes AND, OR, and NOT operators, we can represent $Q$ as multiple conjunctive queries, union-ed with OR. That is, query $Q$ can be transformed to a query of the form $Q_1 OR Q_2 OR ... OR Q_n$. We can therefore represent query $Q$ as a list of conjunctive queries $[Q_1, Q_2, ..., Q_n]$ where each $Q_i$ includes only AND operators (with the same SELECT and FROM clauses as in query $Q$). This is done by converting $Q$’s WHERE clause into disjunctive normal form (DNF) [DP90, Whi61, DH99], using simple logical transformation rules, and by considering each conjunctive disjunct as a separate query. For simplicity, we refer to this list $([Q_1, Q_2, ..., Q_n])$ as the DNF-list, and denote it as $Q_{1,2,...,n}$.

5.2.2 Second Observation

Consider estimating the cardinality of a general query $Q$, using its representing DNF-list of conjunctive queries $[Q_1, Q_2, ..., Q_n]$. Query $Q$’s cardinality can be calculated by a simple algorithm, as follows$^1$:

- Calculate $a = |Q_1|$.
- Calculate $b = |Q_{2,3,...,n}|$.
- Calculate $c = |Q_1 \cap Q_{2,3,...,n}|$.
- Then, $|Q| = a + b - c$.

Quantity $a$ can be calculated using any cardinality estimation model that supports conjunctive queries. Quantity $b$ is calculated, recursively, using the same algorithm, since the list contains only conjunctive queries, and forms a proper input for the algorithm. Similarly, quantity $c$ is calculated, recursively, as described below:

$$|Q_1 \cap Q_{2,3,...,n}| = |Q_1 \cap [Q_2, Q_3, ..., Q_n]|$$ (5.1)

$^1$We use $|·|$ but we could also have used $||·||$ ($|·|$ and $||·||$ are defined in Section 4.2).
Equation 5.1 is equivalent to Equation 5.2:

\[
\left| \left[ Q_1 \cap Q_2, Q_1 \cap Q_3, \ldots, Q_1 \cap Q_n \right] \right|
\]

Therefore, we use the same algorithm since the resulting list also contains (n-1) conjunctive queries and forms a proper input to the algorithm with fewer queries.

The algorithm is exponential in the size of the DNF-list. Given a query \( Q \) for cardinality estimation, using the GenCrd algorithm, we call the cardinality estimation model at most \( C(m) \) times, where \( m \) is the size of the representing DNF-list:

\[
C(m) = 2^m - 1
\]

Note that \( C(m) \) is an upper bound. As the number of OR operators is usually small, this expression is practically not prohibitive.

```
GenCrdRec(DNF_list):
    if len(DNF_list) == 1:
        if ImplyFalse(DNF_list[0]):
            return 0
        return Cardinality(DNF_list[0])
    else:
        cnj_q = [DNF_list[0]]
        smaller_list = DNF_list[1:end]
        updated_list = [q \cap cnj_q For q in smaller_list ]

        return GenCrdRec(cnj_q)
        + GenCrdRec(smaller_list)
        - GenCrdRec(updated_list)

GenCrd(Q):
    return GenCrdRec(GetDNFlist(Q))
```

Figure 5.1: The GenCrd Algorithm.

In Figure 5.1, \( Q_1 \cap Q_2 \) is the intersection query of \( Q_1 \) and \( Q_2 \), one whose SELECT and FROM clauses are identical to \( Q_1 \)'s (and \( Q_2 \)'s) clauses, and whose WHERE clause is \( Q_1 \)'s AND \( Q_2 \)'s WHERE clauses. Function GetDNFlist returns the list of conjunctive queries representing query \( Q \). Function Cardinality(\( Q \)) can be implemented by using any limited cardinality estimation model for estimating the cardinality of the given input conjunctive query \( Q \) (see a simple example in Section 5.2.4).
As described in Figure 5.1, during the execution of the GenCrd algorithm we create multiple conjunctive queries from multiple smaller queries. These queries may often contain contradictory predicates. Therefore, to reduce the prediction time, we use the ImplyFalse algorithm, before directly using the cardinality estimation models. If ImplyFalse returns True, then the result of query \( Q \) has zero-cardinality, and therefore we do not need to call the cardinality estimation model subroutine. This way, the actual number of times we call the cardinality estimation model is, practically, smaller than the upper bound given in the formula for \( C(m) \).

Depending on its implementation, Function \( \text{Cardinality}(Q) \) returns the cardinality of \( Q \) with, or without, duplicates. In the first (respectively, second) case, GenCrd will therefore return cardinalities with (respectively, without) duplicates. We exhibited a generic method to estimate set-theoretic cardinalities, using PUNQ. Thus, if needed, we can implement function \( \text{Cardinality}(Q) \) such that it returns set-theoretic cardinalities.

### 5.2.3 The ImplyFalse Algorithm

The ImplyFalse algorithm takes as input a conjunctive query \( (Q) \) with equality joins, and checks whether there are any contradictory predicates, if there are, it returns True, otherwise, False.

The ImplyFalse algorithm runs in four main stages, as described in Figure 5.2. (1) It first initializes three maps with single element classes. Initially, each class includes a single column, with initial values accordingly. (2) In the first For-loop, it unions the classes of the columns that must be equal using the function \( \text{UnionClasses}(c1,c2) \) which unions the classes of columns \( c1 \) and \( c2 \) into a single class. Hence, each class includes all the columns that must have equal values. (3) In the second For-loop, it updates the maps according to the columns’ predicates. (4) Finally, in the last loop, it checks whether there are any contradictory predicates.

Determining whether a conjunctive Boolean expression is equivalent to False is known to be a co-NP-complete problem in its full generality [GJ79, Ull89, AHV95]. However, our case is tractable, and the problem is solvable in linear time in the number of columns used in the input query, as described in Figure 5.2. This is due to the form of the examined conjunctive queries (comparison to constants). The examined conjunctive queries include joins of one type only, equality join \( (col1,op,col2) \) where \( op \) is \( = \). In addition, the columns’ predicates are of the form \( (col,op,val) \) where \( op \in [<, =, >] \). \(^2\)

\(^2\)Operator \( \leq \) can be expressed with \( <, = \) and OR. This holds for operator \( \geq \) as well.
In Figure 5.2, we first initialize the maps \( \text{minVals}, \text{maxVals} \) and \( \text{exactVals} \) with single element classes, each including a different column, with initial values accordingly. In the first loop, we union classes of columns that must have the same values, using the \( \text{UnionClasses}(c1,c2) \) function which unites the classes of columns \( c1 \) and \( c2 \) into a single class in all the maps. For example, suppose map \( m \) is:

\[
\text{m} = \{ [c1] : \bot, [c2] : \bot, [c3] : \bot, [c4] : \bot, [c5] : \bot \}
\]

Then after calling \( \text{UnionClasses}(c1,c4) \), we get,

\[
\text{m} = \{ [c1,c4] : \bot, [c2] : \bot, [c3] : \bot, [c5] : \bot \}
\]

In addition, the \( \text{Map}[c] \) operator returns the corresponding value of map Map, of the appropriate class in which column \( c \) is located. The \( \infty, \bot \) symbols denote an infinity and uninitialized value, respectively. Functions \( \text{Min}(x,y) \) and \( \text{Max}(x,y) \) return the minimum and the maximum between \( x \) and \( y \), respectively.

**ImplyFalse(Query \( Q \)):**

- \( \text{minVals} = \{ [\text{col}] : -\infty \mid \text{col} \text{ is a column used in } Q \} \)
- \( \text{maxVals} = \{ [\text{col}] : \infty \mid \text{col} \text{ is a column used in } Q \} \)
- \( \text{exactVals} = \{ [\text{col}] : \bot \mid \text{col} \text{ is a column used in } Q \} \)

For-each join clause \((\text{col1},=,\text{col2})\) in \( Q \):

   \( \text{UnionClasses}(\text{col1},\text{col2}) \)

For-each column predicate \((\text{col},\text{op},\text{val})\) in \( Q \):

   if \( \text{op} == '=' \):
      if \( \text{exactVals}[\text{col}] \neq \bot \) and \( \text{exactVals}[\text{col}] \neq \text{val} \):
         return True /* not satisfiable */
      \( \text{exactVals}[\text{col}] = \text{val} \)
   if \( \text{op} == '<' \):
      \( \text{maxVals}[\text{col}] = \text{Min}(\text{val}, \text{maxVals}[\text{col}]) \)
   if \( \text{op} == '>' \):
      \( \text{minVals}[\text{col}] = \text{Max}(\text{val}, \text{minVals}[\text{col}]) \)

For-each \( \text{col} \) in all the columns used in \( Q \):

   if \( \text{maxVals}[\text{col}] \leq \text{minVals}[\text{col}] \):
      return True /* not satisfiable */

   if \( \text{exactVals}[\text{col}] \neq \bot \) and not
      \( \text{minVals}[\text{col}] \leq \text{exactVals}[\text{col}] \leq \text{maxVals}[\text{col}] \):
      return True /* not satisfiable */

return False

Figure 5.2: The ImplyFalse Algorithm.
5.2.4 A simple example

We describe a simple example showing how the GENCRD algorithm functions on general query input.

Given general SQL query $Q$ as follows:

\[
Q: \text{SELECT *}
\text{FROM title t, movie_companies mc}
\text{WHERE t.id = mc.movie_id AND [}
\text{(mc.company_id = 8 AND NOT t.production_year <= 2013)}
\text{OR t.production_year = 2010}
\text{OR t.production_year = 2000}
\text{]}
\]

The GENCRD algorithm first converts general query $Q$ with AND, OR, and NOT operators into a DNF form, creating a list of conjunctive queries [$Q_1, Q_2, Q_3$], as described below.

\[
Q_1: \text{SELECT *}
\text{FROM title t, movie_companies mc}
\text{WHERE t.id = mc.movie_id AND}
\text{mc.company_id = 8 AND t.production_year > 2013}
\]

\[
Q_2: \text{SELECT *}
\text{FROM title t, movie_companies mc}
\text{WHERE t.id = mc.movie_id AND}
\text{t.production_year = 2010}
\]

\[
Q_3: \text{SELECT *}
\text{FROM title t, movie_companies mc}
\text{WHERE t.id = mc.movie_id AND}
\text{t.production_year = 2000}
\]

Query $Q$’s cardinality is then calculated using the GenCrd algorithm with the DNF-list as input, according to the following equation.

\[
|Q| = |Q_1| + |Q_2| - |Q_1 \cap Q_2| + |Q_3| - |Q_1 \cap Q_3| - |Q_2 \cap Q_3| + |Q_1 \cap Q_2 \cap Q_3|
\]

Note that the starred queries imply False, (hence, they have zero-cardinality) and they are detected using the ImplyFalse algorithm.
5.3 GenCrd Evaluation

In this section, we examine how well the GenCrd algorithm transforms limited cardinality estimation models that only handle conjunctive queries into ones that support queries with the AND, OR, and NOT operators\(^3\). To do so, we extend the same four models that were used in the PUNQ evaluation, as described in Section 4.4. Each model was extended by simply using the model in the GenCrd algorithm as a subroutine, implementing the Cardinality function used in the algorithm, see Figure 5.1. For clarity, we denote the GenCrd extended model of \(M\) as \(\text{GenCrd}(M)\).

We evaluate each of the four examined limited models over two test workloads. In addition, we compare our results with those of the PostgreSQL version 11 cardinality estimation component [DBM], as PostgreSQL supports all SQL queries. To estimate cardinality of the examined queries using PostgreSQL, we use the ANALYZE command to obtain the estimated cardinalities.

5.3.1 Evaluation Workloads

The evaluation uses the IMDb dataset, over two different query workloads, GenCrd_test1 and GenCrd_test2. Each workload includes 450 queries that were generated by the same queries generator described in Section 4.3.1 (using a different random seed). Both workloads are equally distributed in the number of joins. From these workloads we generated \(\text{DNF-lists}\), each representing a general query. For each number of joins, the queries are uniformly distributed in terms of the size of the representing \(\text{DNF-list}\), from 1 to 5. That is, each query has representing \(\text{DNF-lists}\) of sizes 1 (conjunctive query without any OR and NOT operators), up to 5 (general query whose representing \(\text{DNF-list}\) includes 5 such conjunctive queries).

<table>
<thead>
<tr>
<th>number of joins</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>GenCrd_test1</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>450</td>
</tr>
<tr>
<td>GenCrd_test2</td>
<td>75</td>
<td>75</td>
<td>75</td>
<td>75</td>
<td>75</td>
<td>75</td>
<td>450</td>
</tr>
</tbody>
</table>

Table 5.1: Distribution of joins. For each number of joins the representing \(\text{DNF-list}\) size is equally distributed from 1 to 5.

\(^3\)Theoretically, the GenCrd algorithm can handle any SQL query with any combination of the AND, OR and NOT operators. However, the examined limited models that are used as subroutines in GenCrd support equality joins only. Thus, in this case, we cannot support queries that apply the NOT operator on equality joins such as \(\text{NOT}(c1 = c2)\), since this join is translated into inequality join \(c1 \neq c2\) (which is not supported in the examined limited models).
5.3.2 The Quality of Estimates

We examined the GenCrd\_test1 workload using two limited models, MSCN and CRN (namely the Cnt2Crd(CRN) model). Recall that this workload includes queries with up to two joins, as MSCN and CRN were trained over such conjunctive queries.

Although the MSCN and CRN models were initially tailored to estimate cardinalities for conjunctive queries only, examining the results in Figure 5.3 and Table 5.2, it is apparent that these models are successfully extended to estimate cardinalities accurately for general queries using the GenCrd algorithm.

![Figure 5.3: Cardinality estimation q-errors on the GenCrd\_test1 workload.](image)

<table>
<thead>
<tr>
<th></th>
<th>50th</th>
<th>75th</th>
<th>90th</th>
<th>95th</th>
<th>99th</th>
<th>max</th>
<th>mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL</td>
<td>1.32</td>
<td>2.57</td>
<td>7.42</td>
<td>16.57</td>
<td>154</td>
<td>52438</td>
<td>132</td>
</tr>
<tr>
<td>GenCrd(MSCN)</td>
<td>2.01</td>
<td>3.59</td>
<td>7.68</td>
<td>11.97</td>
<td>94.12</td>
<td>92684</td>
<td>232</td>
</tr>
<tr>
<td>GenCrd(CRN)</td>
<td>1.68</td>
<td>2.83</td>
<td>6.45</td>
<td>10.65</td>
<td>30.43</td>
<td>538</td>
<td>4.83</td>
</tr>
</tbody>
</table>

Table 5.2: Cardinality estimation q-errors on the GenCrd\_test1 workload.

Note that the GenCrd algorithm is an "analytic" algorithm, therefore it does not include any training, in contrast to other models we presented. Therefore, when using the GenCrd algorithm, the number of joins, in queries in the DNF-list, has no direct effect on result accuracy. The quality of estimates merely depends on the quality of the limited cardinality estimation models (that are used as subroutines in the GenCrd algorithm). In contrast, the PUNQ model estimations, are intrinsically affected by the number of joins, and it over-estimates as the number of joins increases. This is also the case for other machine learning based models that we presented.
5.3.3 Generalizing to Additional Joins

Here we examine the GenCrd algorithm with queries with more than two joins. To do so, we use the GenCrd_test2 workload which includes queries with zero to five joins.

As can be seen in Figure 5.4, queries with 3 joins and more have poorer estimates, as compared with the estimates of queries with zero to two joins. This decline in quality is not directly due to the GenCrd algorithm. The decline stems from training the MSCN and CRN models over conjunctive queries that have only up to two joins. Thus, when MSCN and CRN are used as GenCrd subroutines, DNF-list queries with more joins are not estimated accurately by the original MSCN, or CRN, models.

However, comparing, the errors magnitude when using these limited model \( M \) for cardinality estimation with duplicates (Chapter 3) and the results obtained after extending them using GenCrd algorithm, it appears that there is no significant change in the model’s overall quality. That is, GenCrd maintains the same quality of estimates as the original model \( M \)’s quality of estimates, when \( M \) is used as subroutine. This is expected, since the GenCrd\( (M) \) algorithm obtains an estimation for a given query \( Q \) by simply summing and subtracting several cardinality estimates (see example in Section 5.2.4), where each of these estimates is obtained using the limited model \( (M) \).

![Figure 5.4: Cardinality estimation q-errors on the GenCrd_test2 workload.](image)

<table>
<thead>
<tr>
<th></th>
<th>50th</th>
<th>75th</th>
<th>90th</th>
<th>95th</th>
<th>99th</th>
<th>max</th>
<th>mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL</td>
<td>8.57</td>
<td>168</td>
<td>3139</td>
<td>12378</td>
<td>316826</td>
<td>647815</td>
<td>8811</td>
</tr>
<tr>
<td>GenCrd(MSCN)</td>
<td>4.17</td>
<td>84.92</td>
<td>1887</td>
<td>6769</td>
<td>60405</td>
<td>278050</td>
<td>2611</td>
</tr>
<tr>
<td>GenCrd(Impr. Post.)</td>
<td>2.18</td>
<td>10.97</td>
<td>82.75</td>
<td>286</td>
<td>2363</td>
<td>162894</td>
<td>750</td>
</tr>
<tr>
<td>GenCrd(Impr. MSCN)</td>
<td>2.89</td>
<td>8.45</td>
<td>27.1</td>
<td>73.59</td>
<td>537</td>
<td>5183</td>
<td>38.43</td>
</tr>
<tr>
<td>GenCrd(CRN)</td>
<td>2.26</td>
<td>6.03</td>
<td>17.49</td>
<td>71.17</td>
<td>632</td>
<td>6025</td>
<td>47.24</td>
</tr>
</tbody>
</table>

Table 5.3: Cardinality estimation q-errors on the GenCrd_test2 workload.
Table 5.3 and Figure 5.4 display experimental results over the GenCrd\_test2 workload. The GenCrd(CRN) model is significantly more robust in generalizing to queries with additional joins compared to other model extensions. The GenCrd(CRN) model reduces the mean q-error by a factor x55 compared with GenCrd(MSCN) and by a factor of x185 compared with PostgreSQL. Recall that all the limited models were initially tailored to estimate cardinalities for conjunctive queries only. Hence, these results highlight the usefulness of the GenCrd method in extending limited models to handle AND, OR, and NOT.

### 5.4 Cardinality Prediction Time

Examining the results of the experiments in Section 5.3, we find that the size of the DNF-list does not affect the quality of the estimates. That is, queries with representing DNF-lists of different sizes, are estimated similarly as long as their queries have the same number of joins. However, the DNF-list size directly affects the prediction time. The larger the DNF-list is, the larger the calculation time is.

Table 5.4 depicts the average prediction time in milliseconds, for estimating the cardinality of a single query, when examining different models, with different DNF-list sizes. Note that the PostgreSQL prediction time is not affected by the size of the DNF-list, as it does not use GenCrd.

<table>
<thead>
<tr>
<th>Model</th>
<th>DNF – list size</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL</td>
<td></td>
<td>1.75</td>
<td>1.75</td>
<td>1.75</td>
<td>1.75</td>
<td>1.75</td>
</tr>
<tr>
<td>GenCrd(MSCN)</td>
<td></td>
<td>0.5</td>
<td>1.1</td>
<td>1.55</td>
<td>2.25</td>
<td>2.78</td>
</tr>
<tr>
<td>GenCrd(Imp. Post.)</td>
<td></td>
<td>70</td>
<td>166</td>
<td>261</td>
<td>374</td>
<td>541</td>
</tr>
<tr>
<td>GenCrd(Imp. MSCN)</td>
<td></td>
<td>35</td>
<td>77</td>
<td>120</td>
<td>186</td>
<td>254</td>
</tr>
<tr>
<td>GenCrd(CRN)</td>
<td></td>
<td>16</td>
<td>33</td>
<td>53</td>
<td>72</td>
<td>105</td>
</tr>
</tbody>
</table>

Table 5.4: Average prediction time in ms of a single query with GenCrd algorithm.

Observe that queries with DNF-list of size 1 are conjunctive queries, and therefore their prediction time is equivalent to those described in Table 3.13 in Section 3.4. Despite the prediction time increase, time is still in the order of milliseconds\(^4\). Furthermore, the GenCrd algorithm can easily be parallelized by estimating the cardinality of all the sub-queries in parallel. I.e., by paralleling all the calls done to the limited model. The estimation times described in Table 5.4 are based on the sequential version of GenCrd.

\(^4\)On average, over the GenCrd\_test2 workload’s queries, GenCrd(CRN) model prediction time for a single query is 55 ms, while the actual query’s execution time is 240000 ms (4 minutes).
Chapter 6

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 [CM77] 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 [Ull89]. This also holds in additional settings involving inclusion and functional dependencies [Ull89, AHV95, JK84].

Although determining whether query $Q_1$ is contained in query $Q_2$ (analytically) in the case of conjunctive queries is an intractable problem in its full generality, there are many tractable cases. For instance, in [Sar91, Sar90] 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 $Q_1$. In [CR00] it was proved that for every $k \geq 1$, conjunctive query containment could be solved in polynomial time, if $Q_2$ has query width\(^1\) smaller than $k + 1$. In addition to the mentioned cases, there are many other tractable cases [RBdS+13, Cal06, Cha92, FNTU07]. Such cases result from imposing syntactic or structural restrictions on the input queries $Q_1$ and $Q_2$.

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 work, we address this problem using ML techniques.

\(^1\)The query width of query $Q$ is the minimum tree width over all its query decompositions. Each query decomposition is a tree as defined in [CR00].
Beyond the class of conjunctive queries, queries that include the AND, OR, and NOT operators in their WHERE clauses constitute a broad class of frequently used queries. Their expressive power is equivalent to that of a large fragment of the relational algebra. Therefore, this class of queries had also been extensively researched early on by the database theory community. Sagiv and Yannakakis [SY80] showed that testing equivalence of relational expressions with the operators select, project, join, and union is complete for \( \Pi^p_2 \) of the polynomial-time hierarchy. Chandra and Merlin [CM77] showed that determining containment of conjunctive queries is an NP-complete problem. This also holds under additional settings [Ull89, AHV95, JK84]. Estimating cardinalities of such queries was also intensely researched early on, due to its implications for query optimization [SAC+79].

Lately, we have witnessed extensive adoption of machine learning, and deep neural networks especially, in many different areas and systems, and in particular in databases. Recent research investigates machine learning for classical database problems such as join ordering [MP18], index structures [KBC+18], query optimization [KYG+18, OBGK18], and concurrency control [BS19]. In our context, machine learning was also proposed as a promising approach for solving the cardinality estimation problem [KKR+19, WHT+19].

In addition to the recently proposed machine learning based techniques, there were many attempts to tackle the problem of cardinality estimation. The principal approaches were database profiling [DBM], handling histograms [AC99, BCG01], and Random Sampling techniques [BPP15, OR90, LRG+17] as will be explained subsequently.

Database profiling techniques are based on maintaining, during database updates, statistical information about numbers and sizes of tuples, and distribution of attribute values for base relations, as part of the database meta information. Given a query, the optimizer tries to estimate cardinality by calculating, using these parameters, within a (simple) statistical model during query optimization. Typically, the statistical model is based on the uniformity and independence assumptions of values distribution, both are typically not valid in practice, but they allow for simple calculations, which in turn result in limited accuracy.

Histograms based techniques are based on maintaining static or dynamic histograms for base relations columns, as part of the database meta information. This approach was used in order to tackle the deficiency of the questionable uniformity and independence assumptions of database profiling techniques. Given a query, the optimizer tries to estimate its cardinality according to the information obtained from the relevant histograms within a statistical model.
The sampling techniques [BPP15, OR90, LRG+17] are based on maintaining multiple samples for each table in the database, as part of the database meta information. Given a query, the optimizer tries to estimate its cardinality by running the query on some relevant samples, and then extrapolates the result to the full database.

In the deep learning context, the MSCN model [KKR+19] was recently proposed to solve the cardinality estimation problem. The MSCN model is a sophisticated NN tailored for representing conjunctive SQL queries and predicting cardinalities. Technically, it ingeniously presents queries that vary in their structure to a single fixed NN. MSCN has been shown to be superior in estimating cardinalities for conjunctive 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 than two joins (it is not trained on such queries).

Despite the variety of methods and techniques that were proposed to solve the problem of cardinality estimation, including the recently proposed deep learning based techniques, all these attempts have addressed, conceptually, the problem directly in the same way. The problem was addressed as a monolithic box, where the input is a query, and the output is the cardinality estimate. In this work, we address this problem differently by using information about queries that have already been executed in the database (the query pool).

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 [AC99, BCG01, CR94, KD98]. Using the query pool, we only use the actual cardinalities of queries that have already been executed in the database. Using this information along with the predicted containment rates between the already executed queries and the new examined query, we obtain accurate cardinality estimation.
Chapter 7

Conclusions and Future Research Plans

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. 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 models.

We introduced a novel approach for cardinality estimation, based on the CRN-based containment rate estimation model, and with the help of a query pool. We showed the superiority of our 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 without the need to change the model itself, by embedding it within a three step method. 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.

To make our approach more applicable, as well as other limited models that can only estimate cardinalities with duplicates, we introduced the PUNQ model. The PUNQ model is tailored for estimating the ratio between the set-theoretic cardinality and the cardinality with duplicates.

Another deficiency of many predictors, is the lack of support for general queries that employ combinations of the AND, OR and NOT operators. To overcome this deficiency without altering existing limited models (that handle only conjunctive queries), we introduce a recursive algorithm, GenCrd. GenCrd provides estimates
for queries employing the AND, OR and NOT operators. GenCrd uses existing models, unaltered, as subroutines.

As a result, PUNQ and GenCrd provide a uniform extension, of any restricted estimator, into one supporting general queries and that can also provide estimates without duplicates. For both extensions, providing set-theoretic estimates and estimating general queries, we conducted extensive experimentation. The experiments show that both extensions are accurate and highly practical. Thus, we believe that these extension methods are highly promising and practical for solving the cardinality estimation problem in a fuller generality.

Interestingly, with the extensions we presented, we can estimate cardinalities of queries with GROUP BY (the cardinality is that of the same query with a modified SELECT CLAUSE, i.e., selecting on DISTINCT grouping attributes). In the future, we plan to generalize our extensions to queries with a HAVING clause as well as ones using EXISTS. Also, we plan to focus on estimating cardinalities of nested SQL queries.

Possible Extensions

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

Strings. A simple addition to our current implementation of the CRN and the PUNQ model may support equality predicates on strings. To do so, we could hash all the possible string literals in the database into the integer domain. 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 and the PUNQ model. To support such predicates we need to change the model architecture. Note that predicates such as BETWEEN and IN, may be converted to ordinary predicates.

Inequality joins. In the CRN and the PUNQ models, we only supported equality joins between two columns ($c_1 = c_2$). Supporting inequality joins (e.g., $c_1 > c_2$) requires configuring the model’s vectors segmentation. To support such joins, we may add a new segment along with the J1-seg and J2-seg segments used to represent the joined columns, to represent the join operator ($<, = or >$).
Alternatively, we can keep the J1-seg and J2-seg segments without any changes, for representing equality joins only. For representing inequality joins (such as $c_1 > c_2$ or $c_3 < c_4$), we may add two new segments IJ1-seg and IJ2-seg for representing inequality joins columns ($c_1$ and $c_4$ are represented in IJ1-seg segment, and $c_2$ and $c_3$ are represented in IJ2-seg segment).

**EXCEPT Operator.** Given a query $Q$ of the form $Q_1$ EXCEPT $Q_2$. Our techniques can estimate cardinalities (with or without duplicates) for such query, as follows:

$$|Q_1 \text{ EXCEPT } Q_2| = |Q_1| - |Q_1 \cap Q_2|$$

$$||Q_1 \text{ EXCEPT } Q_2|| = ||Q_1|| - ||Q_1 \cap Q_2||$$

The cardinality (with or without duplicates) of queries $Q_1$ and $Q_1 \cap Q_2$, can be estimated using our techniques.

**Union Operator.** Given a query $Q$ of the from $Q_1$ UNION $Q_2$. Our techniques can estimate cardinalities (with or without duplicates) for such query, as follows:

$$|Q_1 \text{ UNION } Q_2| = |Q_1| + |Q_2|$$

$$||Q_1 \text{ UNION } Q_2|| = ||Q_1|| + ||Q_2|| - ||Q_1 \cap Q_2||$$

The cardinality (with or without duplicates) of queries $Q_1$, $Q_2$ and $Q_1 \cap Q_2$, can be estimated using our techniques.

**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 and the PUNQ models with a new updated training set. This comes with a considerable compute cost for re-executing queries to obtain up-to-date training set samples 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 (CRN or PUNQ) 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. Setting, in advance, a high limit on the maximum value of each column seems most appropriate for this deficiency.

Note that GenCrd is an analytic algorithm, which is not based on a specific database. Thus, unlike the CRN and PUNQ models, the GenCrd algorithm is not directly affected by changes that occur in the database. However, the models that are used within it as subroutines (e.g., MSCN or CRN), are affected by changes in the database. Therefore, in order to ensure accurate estimates when using the GenCrd method, the models that are used as subroutines in GenCrd need be up-to-date.
Appendix A

Appendix

A.1 Box and Whiskers Plot

As described in Wikipedia [Wika], a box plot, or boxplot is a method for graphically depicting groups of numerical data through their quartiles. Box plots may also have lines extending vertically from the boxes (whiskers) indicating variability outside the upper and lower quartiles, hence the term box-and-whisker plot. Outliers may also be plotted as individual points. The spacings between the different parts of the box indicate the degree of dispersion (spread) and skewness in the data and also show outliers. Box plots can be drawn either horizontally or vertically; they received their name from the box in the middle.

In order to display the q-error results on the evaluated tests in this work, we use this kind of plots to display the results in a readable and visual way. Note that according to the graph anatomy, the more skewed the box and the whiskers are towards the q-error 1, the better the results are, as the majority of the test q-error results are close to the q-error 1, which is the best q-error.

All the the presented box plots in this work consist of five lines, as described in Figure A.1. All figures scales are logarithmic with base 10. We distinguish between the over-estimated and the under-estimated samples, their corresponding q-errors are placed above/on right and below/on left q-error 1 respectively.

Figure A.1: Box and Whisker Plot.
A.2 The Challenging IMDB database

Many research papers on query processing and optimization use standard benchmarks like TPC-H, TPC-DS, or the Star Schema Benchmark (SSB) [TPC, OOC09]. While these benchmarks have proven their value for evaluating query engines, [LGM+15] shows that they are not good benchmarks for the cardinality estimation component of query optimizers. The reason is that in order to easily be able to scale the benchmark data, the data generators are using the very same simplifying assumptions (uniformity, independence, principle of inclusion) that query optimizers make. Real-world data sets, in contrast, are full of correlations and non-uniform data distributions, which makes cardinality estimation much harder. Therefore, in [LGM+15], instead of using a synthetic data set, they used the Internet Movie Data Base (IMDB) that contains a plethora of information about movies and related facts about actors, directors, production companies, etc. Moreover, they have shown that the cardinality estimation on this dataset is challenging, while on standard benchmarks like TPC-H it is a trivial task. This is illustrated in Figure A.2. The Figure shows the PostgreSQL cardinality estimates for 4 JOB test workloads on the IMDb database, and 3 test workloads on the TPC-H database. It is clear that the cardinality estimation in IMDB database is challenging, while in TPC-H it is a trivial task as all the box plots are on the q-error value 1.

![Figure A.2: PostgreSQL cardinality estimates for different test workloads on the IMDb and the TPC-H databases.](image-url)
A.3 More Accurate Cardinality Estimates Lead to Better Query Plans

As explained in [LGM+15], given a query to be executed in the database, to obtain an efficient query plan, the query optimizer enumerates some subset of the possible join orders. Using cardinality estimates as a principal input, the cost model then chooses the cheapest alternative from semantically equivalent plan alternatives. Theoretically, if all possible plans are examined, as long as the cardinality estimations and the cost model are accurate, this architecture obtains the optimal query plan. In reality, not all plans are examined and cardinality estimates are usually computed based on simplifying assumptions like uniformity and independence. In real world data sets, these assumptions are frequently wrong, which may lead to sub-optimal and sometimes disastrous plans.

To measure the impact of cardinality misestimation on query performance, the authors of [LGM+15] conducted the following experiment.

They tested a query workload in the PostgreSQL DBMS, while injecting to it cardinality estimates that are obtained from different cardinality estimation methods and then executed the resulting plans. They injected estimates that are obtained from 5 different cardinality estimation methods: DBMS A, DBMS B, DBMS C, HyPer, and the PostgreSQL cardinality estimation component.

Additionally, they inject the true cardinalities, which computes the optimal plan (with respect to the cost model). Table A.1 describes the run-times based on their slowdown w.r.t. the optimal plan (obtained when using true cardinalities). The run-times slowdown are grouped in different domains (each in different column) as described in the table. When using the estimates instead of the true cardinalities, the first column describes the percentage of queries that become slightly faster. The second column describes the percentage of queries whose run time have almost not changed. The third till the last column describe the percentage of queries that become slower.

<table>
<thead>
<tr>
<th></th>
<th>&lt; 0.9</th>
<th>[0.9, 1.1)</th>
<th>[1.1, 2)</th>
<th>[2, 10)</th>
<th>[10, 100)</th>
<th>&gt; 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL</td>
<td>1.8%</td>
<td>38%</td>
<td>25%</td>
<td>25%</td>
<td>5.3%</td>
<td>5.3%</td>
</tr>
<tr>
<td>DBMS A</td>
<td>2.7%</td>
<td>54%</td>
<td>21%</td>
<td>14%</td>
<td>0.9%</td>
<td>7.1%</td>
</tr>
<tr>
<td>DBMS B</td>
<td>0.9%</td>
<td>35%</td>
<td>18%</td>
<td>15%</td>
<td>7.1%</td>
<td>25%</td>
</tr>
<tr>
<td>DBMS C</td>
<td>1.8%</td>
<td>38%</td>
<td>35%</td>
<td>13%</td>
<td>7.1%</td>
<td>5.3%</td>
</tr>
<tr>
<td>HyPer</td>
<td>2.7%</td>
<td>37%</td>
<td>27%</td>
<td>19%</td>
<td>8.0%</td>
<td>6.2%</td>
</tr>
</tbody>
</table>

Table A.1: Run-time slowdown when using estimates instead of true cardinalities.
A small number of queries become slightly faster when using the estimated instead of the true cardinalities. This effect is caused by cost model errors and not by the cardinality estimation errors.

However, the vast majority of the queries are slower when estimates are used instead of the true cardinalities. For example, using DBMS B’s model cardinality estimates, only 0.9% of the queries become faster when using the estimates instead of the true cardinalities. For 35% of the queries, the run time has almost not changed. However, 65.1% of the queries are slower when estimates are used instead of the true cardinalities. In particular, 18% of the queries are 1.1x up to 2x slower. 15%, are 2x up to 10x slower. 7.1% are 10x up to 100x slower, and 25% of the queries are more than 100x slower (when using estimates).

This experiment shows that all cardinality estimators occasionally lead to plans that take too long to execute, and illustrates the fact that when the cardinality estimates are more accurate, the chosen query plans are better, and as a result the queries execution run times are usually shorter.

### A.4 CRN and PUNQ implementation

In both the CRN and PUNQ models, each query $Q$ was first represented using set $V$ of vectors. Each vector was presented to the neural network ($MLP_1$ or $MLP_2$ in CRN, and $MLP_{mid}$ in PUNQ) separately, obtaining a vector of size $H$ as a result. To do so, each query was represented using a matrix of size $NxL$, where $N$ is the number of vectors that represent the query (the size of set $V$), and $L$ is the vector’s dimension. This matrix was presented to the neural network, and as a result, a matrix of size $NxH$ was obtained (obtaining $N$ vectors, each of size $H$, that were then averaged, creating the representative vector $Qvec$).

In practice, we implement a vectorized version of our models that operates on mini-batches of data of size $B$. As a result each mini-batch was represented as a matrix of size $BxNxL$. As the number of set elements in each data sample in a mini-batch can vary, we pad all samples with zero-valued feature vectors that act as a dummy set of elements so that all samples within a mini-batch have the same number of set elements ($N$). That is, we used the maximal $N$ value, so that each query on the mini-batch has at most $N$ "real" vectors in its representing set $V$ of vectors (with the added dummy vectors, each query has exactly $N$ vectors). Subsequently, we mask out dummy set elements in the averaging operation (for creating $Qvec$) so that only the original (real) set elements contribute to the average for each query of the $B$ queries of the mini-batch.
Both models (CRN and PUNQ) were trained using the efficient Adam optimizer. The Adam optimizer is an efficient gradient descent based algorithm. In particular, in both models, the learned weights were updated using the backpropagation algorithm. The backpropagation algorithm aims to update each of the weights in the network so that they cause the actual output to be closer to the target output, thereby minimizing the prediction error according to the defined loss function. In the basic gradient descent algorithms, this is done as described below [Maz].

Assuming that a NN-based model predicts \( \hat{y} \) as output, given the input, weights and biases of the model’s current state. Further, assume that the model’s loss function is denoted as \( Loss(y, \hat{y}) \), where \( y \) is the actual output. For each weight \( w_i \) of the model’s neural network, the backpropagation algorithm updates the weights according to the equation below:

\[
\hat{w}_i = w_i - \eta \frac{\partial Loss(y, \hat{y})}{\partial w_i}
\]

Where \( \frac{\partial Loss(y, \hat{y})}{\partial w_i} \) is the partial derivative of \( Loss(y, \hat{y}) \) with respect to \( w_i \), and \( \eta \) is the learning rate. \( \frac{\partial Loss(y, \hat{y})}{\partial w_i} \) reflects how much a change in \( w_i \) affects the total error. Observe that the partial derivative is calculated by applying the chain rule along the path from the model’s output back to the weight \( w_i \) (see Figure A.3).

![Figure A.3: A simple example, illustrating how the partial derivative of \( E_{total} \) with respect to \( w_1 \) is calculated, using the chain rule (\( E_{total} \) is the Loss function).](image-url)
A similar procedure is done for all the PUNQ and the CRN models’ weights during training. In both the CRN and the PUNQ models there exists an average layer (the AVG layer) that outputs the representative vector $Qvec$ (obtained by averaging the output vectors obtained from the previous neural network). Therefore, given a weight $w_i$ that we aim to update using the backpropagation algorithm, we distinguish between two cases:

- The weight $w_i$ is located in a layer after the average layer.
- The weight $w_i$ is located in a layer before the average layer.

**The weight is located in a layer after the average layer**

In such cases, the backpropagation is done regularly as described above since the path from the output back to the weight $w_i$ does not include the average layer. That is, the backpropagation algorithm updates the weight according to the same equation:

$$
\hat{w}_i = w_i - \eta \frac{\partial \text{Loss}(y, \hat{y})}{\partial w_i}
$$

Here, the partial derivative is calculated using the chain rule (without passing through the average layer).

**The weight is located in a layer before the average layer**

In such cases, assuming that the input query is represented using $N$ "real" input vectors $(V_1, V_2, ..., V_N)$, the backpropagation algorithm updates the weight according to the following equation:

$$
\hat{w}_i = w_i - \eta \frac{1}{N} \sum_{j=1}^{N} \frac{\partial \text{Loss}(y, \hat{y})}{\partial w_i}(j)
$$

Where $\frac{\partial \text{Loss}(y, \hat{y})}{\partial w_i}(j)$ is the partial derivative of $\text{Loss}(y, \hat{y})$ with respect to $w_i$, when considering the input vector $V_j$. That is, the partial derivative is calculated according to the values when the input vector is vector $V_j$.

Recall that when training in mini-batches, dummy vectors may exists, however, these vectors are ignored in the average layer.
Bibliography


Intelligence Techniques for Data Management, aiDM@SIGMOD. ACM, 2019.


[DBM] PostgreSQL DBMS. Postgresql, the world’s most advanced open source relational database. https://www.postgresql.org/.


[LRG+18] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. Query optimization through the looking glass, and what


The algorithm is executed in three steps. In the first step, it builds a set of vectors that are described above. In the second step, it generates a unique vector for the set that is represented by the query. In the third step, it calculates the total time between the two queries based on the vectors. Consequently, a useful, fast, and small, efficient algorithm is obtained that calculates the total time between two queries accurately.

When using the algorithm of Nolchizov we could introduce a new method to solve the query-giving problem of three queries. This method is based on the fact that the first query is among the three queries in the database. In this way, we can use the algorithm of Nolchizov with the results of the first query and the second query to find the time of the third query. This time is calculated using the first and second queries and the third query.

After the implementation of this method and the comparison with known methods for query-giving, it was found that the new method is much more accurate, especially in cases where (within the queries). Previous articles have shown that there is a large number of combinations where as the number of combinations increases, the calculations become less accurate. The success of SubRoutines in handling complex queries (and others) is due to the fact that SubRoutines are represented by queries that are selective and not subject to restrictions.

However, other methods in addition to the algorithm of Nolchizov are not supported (and others) that use recursive methods and selective methods without restrictions. Therefore, the limitations of the method of Nolchizov are overcome.

The first method is based on the use of a deep learning model, which is intended to expand the models used without restrictions (as queries). The second method is based on an algorithm that expands the models used in queries by means of OR and NOT conditions.

In all the experiments performed on several methods of query-giving, the new method was consistently superior to the others in terms of quality. Therefore, the recommendations of the original models are maintained. Accordingly, they do not have a negative impact on the quality of the calculations (as compared with the calculations of the original models).
בכל מערכות ניהול מסדי נתונים, רכיב האופטימיזציה של מעבד ושיאלות נובע מהאומדן מהאומדן, האופטימיזציה של השיאלות בוחן את האפשרויות ובוחר ברווח lại את השיאלה. בבחינת השיאלה, האופטימיזציה בוחן מספר רב של אפשרויות הזרかない. לכל אופטימיזציה מועדת מספר רב של אפשרויות הזרракти שחרר. על כל אפשרויות הזרракти שחרר, על כל מתכון בוחרת הרשויות ייעול של האופטימיזציה למספר נימוקים כל האופטימיזציה למספר נימוקים, כאשר האופטימיזציה יוצר למספר נימוקים כל האופטימיזציה למספר נימוקים. לא מגדיר, כי כל השיאלות עם נימוקי,'#تحويل', ובגרות ההובלה לש Tatto ותרフラטינית.

המחקר מתמקד בהינתן של פורסמו עד דצמבר סמואלי בפובלית לשתי המדעיות.

ילקון מתוחכם בהבורה, הפקולים של המחבר (רוה איציק)

הנמצאת (ODULE שמואלי) במלכד המחקר של המחבר, אשר גרסאותיהם העדכניות

ביוטר העניין:


תודות

אריאט, בברנאי ללביש את התודתי העמוקה והערוכת למנהל של פורסמו

עדכון שומרון על התמיכת הרצפה לאורכן ל獭ויה לנץ, על סבלנות,

המתרגשות והידיעת העצמה שישל. הצלת למשרדי היותת במית פורחנה בכף עשם

שנחקלתי בין כל פייס והשיטתי לי ש البعלה לגב המחקר והכנתה של.

יהיו לי ליבוד עלpaque עם אינט פורסמו סמואלי.

בגון, אני רᓵ להביא את התודתי לכל התורשים של, הבのみות لمدة זמנים.

אנס מעורר מאור את כל התמק שדקשת לה.linkLabel, לא又好 התמיד והמיניות

הצגמהسرائيلית לאורכן על ל獭ויה ולהלך המחקר.

החברה שלל את התיה שלמה ביל התודתי לימדéo בחוק הגוזל בגוזל עזר;

וי, אך יאוחרות. האותב וההתאמהابل מוטניק שערועי, לאراف ל獭ויה

ובאוף לכל באיתס לא יסולם ופפ.
למידת מכונה של מידה התaleza ביני
ﺷאלתות SQL וגודל התוצרת

תיעוד על מחקר

לשם מילוי הלכו של הדרישות לכתוף התואר
מופסק למדעי במדעי המחשב

רוב' האיאט

רותש לתכנית הטכניונ - מכון טכנולוגי לישראל
השון החש"פ חיפה נובמבר 2019
למידת מכונת של מידה ההכלת بين שאילתות \textsc{SQL} והודל החוזאה

רות' חסין