Could we do smarter Cardinality Estimation? Comparing recent Machine Learning approaches

Aparajeeta Jha
Otto-von-Guericke-University
Magdeburg, Germany

Arvindjit Singh
Otto-von-Guericke-University
Magdeburg, Germany

Peer Ahamad Shaik
Otto-von-Guericke-University
Magdeburg, Germany

Pradeep Reddy Kancharla
Otto-von-Guericke-University
Magdeburg, Germany

Sankul Rahul Rathod
Otto-von-Guericke-University
Magdeburg, Germany

Shubham Vivek Manekar
Otto-von-Guericke-University
Magdeburg, Germany

***Abstract*—Query optimization is highly dependent on cardinality estimation. Machine learning and deep learning techniques are emerging as a promising approaches in different database applications and systems. They are being explored for cardinality estimation as well.**

In this paper, we try to implement and replicate two recently proposed ML approaches for cardinality estimation - *MSCN (Multi-set convolutional network)* and *DeepDB*. The main aim is to compare the performances of these two approaches on a common workload - IMDB Dataset and JOBlight benchmark. And hence, find a suitable ML model that can accurately predict cardinalities.

We evaluate our 2 models with different hyperparameter settings based on error values - MSE and MAPE. While MSCN model is robust in capturing correlarions effectively, we find that DeepDB model helps in optimally training the data and provides good estimates for different queries over the database. The data driven, DeepDB model is found to be more interpretable than MSCN.

I. Introduction

"The root of all evil, the Achilles Heel of query optimization, is the estimation of the size of intermediate results, known as cardinalities." - Guy Lohman

Reliable cardinality estimation is one of the key prerequisites for effective cost-based query optimization in database systems. After a query is submitted, the Optimizer follows these five basic steps: Parsing -> Binding -> Optimization -> Execution -> Results. The Cardinality Estimator comes in to picture at the Optimization step which evaluates the execution plans. [1]

In cost-based optimization scenarios, a query is assessed to learn the best and most efficient performance plan on the basis of data estimation. *Cardinalities* refer to the number of tuples or rows processed at each level of a query plan (intermediate results). *Cardinality Estimation* is, thus, a method essential for selecting an optimal query plan. Cardinality Estimator results are heavily influenced by the presence of joins in a query. In the context of databases, a JOIN refers to combining columns from one (self-join) or more tables (multi-join) by using values common to each. JOIN order decides the order or sequence in which the different tables participating in a query are selected and merged. Determining an optimal Join order is significant because if we can join two tables that will minimize the estimate of rows needed to be processed by the consequent steps, then the performance of our query will improve. Selecting a good JOIN order is, thus, helpful in query optimization, optimizing I/O, optimizing buffer usage, reducing the computation, and improving parallel processing. In cost-based optimization scenarios, a query is assessed to learn the best and most efficient performance plan on the basis of data estimation. [2]

Cardinalities are usually estimated by the database using statistics and hard-coded rules. Cardinality estimation is needed to determine cost estimates and the proper join order during query optimization. The query optimizer should be able to make good estimates for intermediate result sizes for it to choose between several plan alternatives(e.g. Fig. **??**). However, several widely-used database systems perform poor estimate calculation resulting in slower queries and unpredictable performance [3].

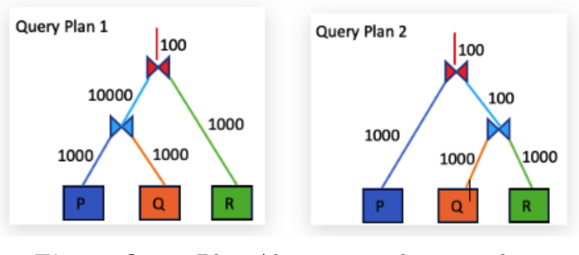

Fig. 1: Query Plan Alternatives depicting how different join order between tables P,Q and R affect the cost(intermediate row results)

The major challenge in cardinality estimation is finding join-crossing correlations. Most traditional approaches for cardinality estimation were statistical approaches that made use of *Histograms* and *Sketches*, which turned out to be hugely problematic in case of a large number of attributes or joins. The other State-of-the-art proposals included *Sampling-Based Approaches*, e.g. Index-Based-Join-Sampling (IBJS) or *Regression-based ML models*. These approaches attempted to address the join correlation problem to an extent but had their drawbacks. Sampling-Based Approaches fail when there are no qualifying samples to start with or when no suitable indexes are available (known as the *0-Tuple problem*) [4] while Regression-based models usually cannot deal with variable-sized inputs (e.g. queries with a varying number of joins).

Why ML for cardinality estimation? Machine learning algorithms serve as a promising solution to the cardinality estimation problem. We can express Cardinality Estimation as a supervised learning problem, with query features denoting the input and the estimated cardinality as the output [4]. ML Models can overcome the challenges faced by traditional approaches - capture correlations and the 0-tuple challenge faced by pure sampling-based approaches. The existing DBMS components have successfully been replaced with learned counterparts (for e.g. learned cost models, learned query optimizers, etc.) using Deep Neural Networks or DNNs [5].

In this paper, we seek to implement two of the ML models for cardinality estimation - MSCN(Multi-set Convolutional Network) proposed by Kipf et. al. (2018) and DeepDB proposed by Hilprecht et. al. (2019). MSCN is a workload-driven approach and extends the traditional sampling-based estimation approaches. MSCN overcomes the challenges faced by sampling-based approaches like a) the 0-tuple problem where the sampled tuples fail to satisfy a predicate, and b) in capturing join-crossing correlations. However, MSCN has two major drawbacks - a) it can be very expensive to collect the training data, and b) training data needs to be recollected with even minor changes in the workload and the data [6]. So apart from Workload-driven approaches, rivaling Database-driven approaches were also proposed. DeepDB is a data-driven model that captures the joint probability distribution of data and reflects correlation among the attributes. There is no need to retrain the model as DeepDB supports direct updates(i.e. inserts, deletes, and updates) on the underlying database [6]. We have selected this work for our study since these models are the leading state-of-the-art solutions for cardinality estimation, and they both should serve as a strong baseline for further research.

Contributions We successfully reproduced the performance improvements achieved by Kipf et. al. using MSCN and by Hilprecht et. al. using DeepDB for estimating the query cardinalities. We implemented our frameworks with different hyperparameter combinations and different workloads to assess the performances of the models. Finally, we drew a comparison of the performance of the models on the common workload - *IMDb dataset* and *JOBlight benchmark*.

Outline The remainder of the paper is structured as follows: In Sec. II, we summarize 3 related works on modern cardinality estimation. With this review, we provide a research context for the work we reproduce in this paper. In Sec. III, we introduce the necessary concepts to understand in detail the prototype design and implementation of the models, which is presented in Sec. IV. With our design, we review the work of Kipf et. al. and Hilprecht et. al. and present the research questions we aim to answer through our study. In Sec. V and Sec. VI, we describe our experimental setup and evaluation. Finally, we conclude in Sec. VII, by summarizing our work and proposing future research directions.

II. Related Work

This section will discuss recent research papers that exploited the topic of cardinality estimation and the different techniques for obtaining high performance and efficient response time for queries in database systems. In particular, the papers selected represent two different approaches for modern cardinality estimation, which are MSCN and DeepDB, next to a paper that critically evaluates benchmarks - these inspire our research work. The first two are workload-driven and data-driven approaches respectively and utilize the Deep Learning concepts and architectures. The last paper focuses more on finding a good join order through different techniques, proposing the JOB benchmark.

A. *Query Optimization Through the Looking Glass, and What We Found Running the Join Order Benchmark*

It is important to find a good join order for improving query performance and resource efficiency. In this paper, the Join Order Benchmark (JOB) is introduced and the classic, traditional cost-based approach for query optimization is used in experimentation on cardinality estimators. The cost model takes cardinality estimates of each plan as input. It then chooses the most inexpensive plan from all available plan alternatives otherwise semantically equivalent. The cardinality estimator plans compute the cardinality with prior assumptions of uniformity and independence. But in real-world data sets, these assumptions fail more often than not which leads to poor results [5].

The classical query optimization architecture is shown in Fig 2.

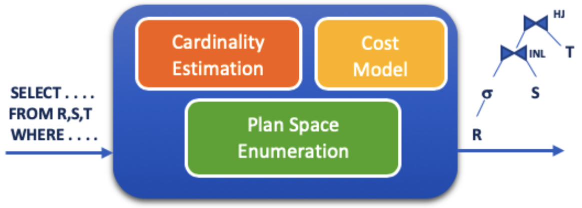

Fig. 2: Traditional query optimizer architecture [3]

Experiments were performed to a) quantifiably evaluate the three main components of the architecture on cardinality estimators, b) the importance of cost-based model accuracy in the overall query optimization process, and c) the size of the enumerated plan space. To achieve this, a novel methodology is used that isolates the impact of different optimizer components on query performance.

The primary contributions of the paper are to design the JOB, based on the IMDB data set. It provides guidelines for the complete end to end design of a query optimizer, using the evaluations of the query optimizer architecture components and providing substantial observations.

The quality of base table cardinality estimates is measured using the *q-error*, which is the difference factor between an estimate and the true cardinality. To measure the estimates for joins, the q-error is calculated for each intermediate result of the query set, distinguishing between over and underestimation.

The experiments for cardinality estimation demonstrate that the systems that retain table samples predict single-table result sizes relatively better than the systems applying the independence assumption and using single-column histograms. Rather than explicitly identifying the correlations between tables, this study emphasizes the robustness of a system towards correlations. This work validates that cardinality estimates deteriorate greatly with an increasing number of joins, if join-crossing correlations are not addressed and are overlooked. This leads to both over and underestimation of result sizes (mostly the latter) [5]. It has also been suggested that tuning cost models provides fewer benefits than improving cardinality estimates.

B. Learned Cardinalities: Estimating Correlated Joins with Deep Learning

This paper by Kipf et. al. [4] introduces a novel deep learning method for estimating cardinalities - Multi-set Convolutional Network or MSCN. In particular, this approach tries to overcome the 0-tuple limitation (a condition when no qualifying samples are present, to begin with, or when no suitable indexes are available) faced by pure sampling-based techniques and to effectively find join correlations in tables. MSCN tries to exploit current machine learning concepts, and expresses cardinality estimation as a supervised learning problem, with query features as input and estimated cardinality as the desired output.

The approach implemented is based on a deep learning model that represents query features as Sets ignoring the different permutations (same cardinality just different expressions) of a query's features. This produces smaller models and better predictions. The central idea is to train a supervised learning algorithm with queries as input and the cardinality as output. The model is then used as an estimator for unseen new queries.

MSCN architecture takes its inspiration from deep learning concepts of DeepSets. The implementation represents a query as Sets of tables, joins, and predicates present in that specific query. A unique one-hot vector represents each table. Joins are also featurised with a unique one-hot encoding. Unique queries with up to two joins were generated. Tables, Joins, and Predicates are represented as separate modules of one two-layer neural network per set component. The query is represented as a collection of multiple sets, which motivates the MSCN model architecture.

For every set S, a set-specific, per-element neural network MLP is learned. Module outputs are averaged, concatenated, and fed into a final output network. All MLP modules used are two-layer fully connected neural networks with $\mathrm{ReLU(x)} = \max(0, \mathrm{x})$ activation functions, unless otherwise stated. For the output MLP, a $\mathrm{sigmoid(x)} = 1/(1 + \exp(\mathrm{x}))$ activation function is used for the last layer to output a scalar $\mathrm{w_{out}} \in [0, 1]$ [4]

The MSCN architecture is shown in Fig 3. The model is trained using Adam Optimizer to minimize the mean q-error. Further explorations are made using mean-squared error and geometric mean q-error.

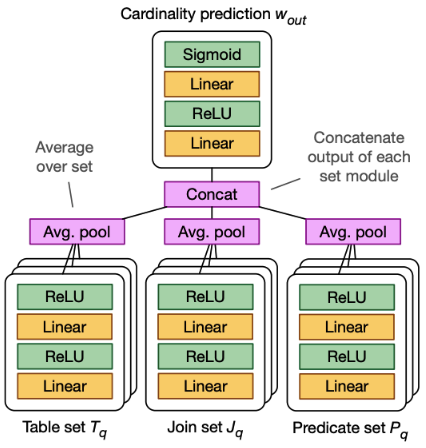

Fig. 3: Architecture of MSCN Model [4]

The model building involved three steps - a) generating random (uniformly distributed) queries, b) executing these queries to find their true cardinalities, and c) feeding this training data into an ML model. The model was implemented on an immutable database snapshot or a read-only database [4]. Increasing the query sample size for training did not increase the prediction time as opposed to the observations for traditional sampling-based approaches.

The approach is evaluated using the IMDB dataset over different query workloads. The q-error results are compared against those produced by using PostgreSQL version 10.3, Random Sampling (RS), and Index-Based Join Sampling (IBJS). The results show that PostgreSQL's errors are skewed and RS tends to underestimate joins. IBJS performed remarkably well in the median but suffers from empty base table samples. Considering the small data used by MSCN as compared to IBJS, MSCN captures (join-crossing) correlations reasonably well and does not suffer as much from 0-tuple situations. MSCN outperforms the competitors and proves to be more robust.

C. *DeepDB: Learn from Data, not from Queries!*

The workload-driven approach, like MSCN, has two major disadvantages - collecting the training data can be very expensive and there is a need to re-train the model whenever the workload or the database changes. To overcome these limitations, a new data-driven approach has been proposed in this paper by Hilprecht et. al. [6], which directly supports changes in the workload and data without the need for retraining. The data-driven approach provides better accuracy than state-of-the-art learned components and also generalizes better to unseen new queries. The main idea of DeepDB is to learn an offline representation of the data. A new class of models called *Relational Sum-Product Networks (RSPNs)* is developed to optimally capture the joint probability distribution over all attributes in a relational database.

The advantages of using RSPNs are multi-fold. RSPNs can be built on arbitrary schema and support complex queries with multi-way joins and aggregation functions (COUNT, SUM, AVG). RSPNs do not need to know join paths a priori and allow true ad-hoc joins. These support direct updates without the need to retrain the model. RSPNs also support NULL value handling and functional dependencies.

Using DeepDB, there is no need to create dedicated training data, i.e. pairs of queries and cardinalities. Since RSPNs capture the characteristics of the data independent of a workload, arbitrary join queries can be supported without the need to train a model for a specific workload. RSPNs are based on Sum-Product Networks which can capture only the data of single tables. SPNs (Sum-product networks) are kind of an index on data, that you can ask queries from, without going back to the data.

RSPNs mitigate the challenges faced by SPNs. For aggregate queries, RSPNs compute probabilities over the variables on the leaves and propagate the values up in the tree. At product nodes, the expectations and probabilities from child nodes are multiplied whereas, on sum nodes, the weighted average is computed. SPNs do not provide solutions for handling NULL values. In RSPNs, NULL values are represented as a dedicated value for both discrete and continuous columns at the leaves during learning. Fig 4 shows the architecture of the DeepDB model used.

RSPNs represent the data as accurately as possible in contrast to SPNs generalizing and approximating the distribution. Functional dependencies between non-key attributes A $\rightarrow$ B are not well captured by SPNs. RSPN resolves this limitation by maintaining a mapping from values of A to values of B in a separate dictionary. The model ignores column B while learning and translates them back accordingly during run-time. Finally, RSPNs extend SPNs by facilitating direct updates of the model independent of the changes in the workload.

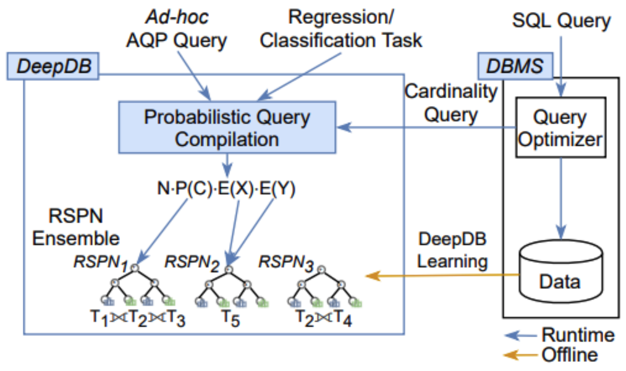

Fig. 4: Architecture of DeepDB Model [6]

The RSPNs used in all experiments were implemented in Python as extensions of SPFlow. As hyperparameters determined using grid-search, an RDC threshold of 0.3, a budget factor of 0.5, and a minimum instance slice of 1% of the input data were used to determine the clustering granularity.

The evaluation is performed on the JOBlight benchmark and synthetic queries. The DeepDB performance was compared with the learned baseline models - a) MSCN, b) an approach based on wavelets built per table, c) Perfect Selectivities approach which uses an oracle to return the true cardinalities for single tables and finally with the non-learned baselines, d) the standard cardinality estimation of Postgres 11.5 and e) online random sampling and Index-Based Join Sampling (IBJS).

For RSPNs, the training time is much lower since training data is not collected for the workload. The DeepDB model experiences a larger storage overhead but the storage-optimized version of DeepDB provides better accuracy than all other baselines. Evaluation based on q-error shows that both DeepDB and the storage-optimized version outperform the other models. IBJS provides a low q-error in the median, and MSCNs outperform traditional approaches for the higher percentiles and prove to be more robust. DeepDB outperforms IBJS in the median and also provides additional robustness, hence outperforming MSCN. The q-errors of both Postgres and random sampling are significantly larger and wavelets have the highest error as they suffer from the curse of dimensionality. Perfect Selectivities provide better errors than wavelets but it is not comparable to DeepDB as it fails to take correlations across tables into account. For uniform/independent data, DeepDB provides no significant advantage over other techniques but outperforms the other baselines for higher degrees of skew/correlation. Comparing generalization over unseen queries, DeepDB performs better than MSCN for larger joins.

To close this section, we note that many approaches exist for cardinality estimation, representing diverse expectations about whether the estimation will be done online or offline, or whether it should be workload-driven or data-driven. In this paper, we focus on comparing both the methods to establish a more robust and correct model. We will also highlight the advantages of one approach over the other in possible real-world scenarios. For future work, it would be interesting to study and compare the other approaches and to formulate an ensemble approach as well.

III. Background

After providing a perspective on the approaches in the field, in this section we introduce the fundamental concepts and background for the techniques that we adopt in our study.

A. DeepSets

The MSCN model architecture is inspired by and extends the concepts of DeepSets which essentially are a neural network module operating on sets. Most traditional approaches in Machine Learning operate on fixed dimensional vectors. DeepSets deal with sets as inputs and especially on objective functions defined on sets invariant to permutations. Fig **??** shows the architecture of an invariant DeepSets model.

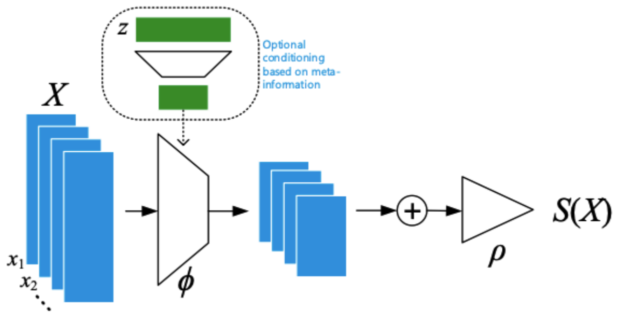

Fig. 5: Architecture of DeepSets: Invariant [7]

The permutation invariant property (i.e, the response of the function does not change with the ordering of the elements)of a function is defined as -

Property 1: A function $f : 2^{\chi} \to \gamma$ acting on sets must be permutation invariant to the order of objects in the set, i.e. for any permutation

$\pi : f(\{x_1, ..., x_{\mathrm{M}}\}) = f(\{x_{\pi(1)}, ..., x_{\pi(M)}\})$ [7]

Where, the input is a set $X = \{x_1, .., x_{\mathrm{M}}\}, x_{\mathrm{M}} \in \chi$, the input domain is the power set $X = 2^{\chi}$, and the function f transforms its domain χ into its range γ. In the supervised setting, given N examples of $\mathrm{X}^{(1)}$,...,$\mathrm{X}^{(\mathrm{N})}$ as well as their labels $\mathrm{y}^{(1)}$,...,$\mathrm{y}^{(\mathrm{N})}$, the task would be to classify/regress (with a variable number of predictors) while being permutation invariant w.r.t. predictors. [7]

A DeepSets structure is based on the below observation by analyzing the invariant case, where χ is a countable set and $\gamma = \mathbb{R}$.

Theorem 1: A function $f(X)$ operating on a set X having elements from a countable universe, is a valid set function, i.e., invariant to the permutation of instances in X, iff it can be decomposed in the form $\rho(\sum_{x \in X} \phi(x))$, for suitable transformations ρ and ϕ . [7]

MSCN is implemented by applying fully-connected multi-layer neural networks (MLPs) to parameterize the functions and . The function approximation of and is used to learn the mappings f(S) for random sets S. [4]

B. Sum-Product Networks or SPNs

Sum-Product Networks (SPNs) form the base concept for Relational Sum-Product Networks(RSPNs). SPNs learn the joint probability distribution $P(X_1, X_2, ..., X_{\mathrm{n}})$ of the variables X_1,X_2,..., X_{n} in the dataset very efficiently [8]. For the sake of our study, Tree-SPNs (trees with sum and product nodes as internal nodes and leaves) are taken into account, as shown in Fig **??**. The sum nodes split the rows of a dataset or the sample population into clusters. The product nodes split the independent variables or the columns of a dataset. Leaf nodes represent a single, independent attribute and their approximate distribution using histograms for discrete data or piece-wise linear functions for continuous data [9].

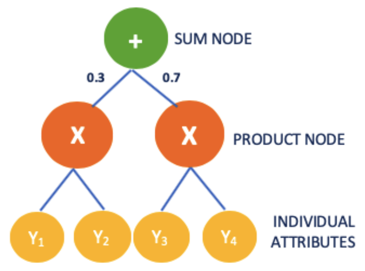

Fig. 6: A simple SPN Tree Structure

SPNs are learned by recursively splitting the data into different clusters of rows (adding a Sum node) or clusters of independent columns (adding a Product node). A standard algorithm like KMeans or a random hyperplane can be used to split the data for the clustering of rows. SPNs, in general, have a polynomial-size and allow inference in linear time w.r.t. the number of nodes [8] [9]. We can estimate the conditional probabilities on random columns. First, the conditions are evaluated on every appropriate leaf and therefore SPNs are evaluated in a bottom-up manner.

IV. Prototype Design and Implementation

A. Project Overview

Fig 7 describes the overview of the workflow for our project. For evaluating the robustness of our framework on different workloads, we trained and tested our models using two different query sets - a small set of queries based on the Join Order Benchmark (JOB and JOBlight) [3] and another larger synthetic query JOB set - based on the

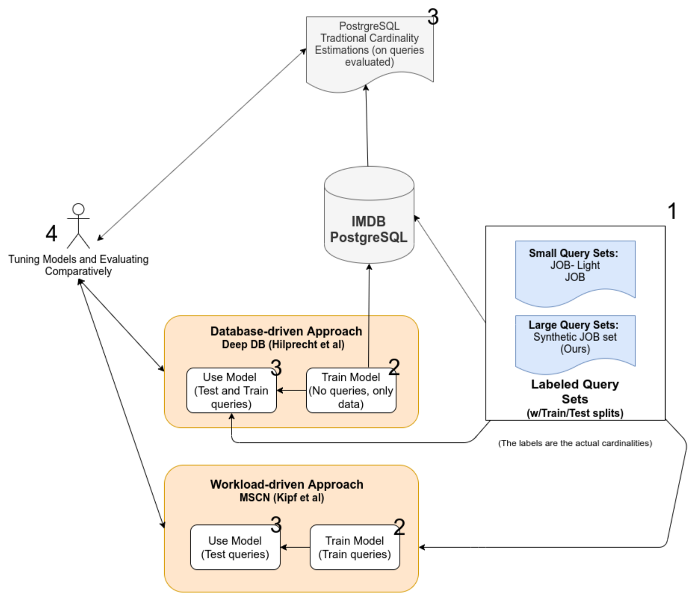
PostrgreSQL Tradtional Cardinality Estimations (on queries evaluated)
3
IMDB PostgreSQL
1
4
Tuning Models and Evaluating Comparatively
Small Query Sets: JOB- Light JOB
Large Query Sets: Synthetic JOB set (Ours)
Labeled Query Sets (w/Train/Test splits)
(The labels are the actual cardinalities)
Database-driven Approach
Deep DB (Hilprecht et al)
Use Model (Test and Train queries)
3
Train Model (No queries, only data)
2
Workload-driven Approach
MSCN (Kipf et al)
Use Model (Test queries)
3
Train Model (Train queries)
2

Fig. 7: Overview of Framework: Comparing And Tuning Modern Cardinality Estimation Approaches

IMDB dataset. The query sets are labeled, with the labels being the actual true cardinalities of the queries, which are then split into training and test sets. Next, we applied the two models - the data-driven approach DeepDB which trains over the entire data [6], unlike the workload-driven approach MSCN which trains on the queries [4]. We followed the same basic architectures for our models as described in the respective papers. We used PostgreSQL to store our database, in a way that can be accessed by both the models. After training both these models, we tested them over the test split on our query sets and predicted the cardinalities. We compared the model outcomes with each other and with PostgreSQL's traditional cardinality estimators over the evaluated queries. Analyzing them over the error measures helped us understand how we can fine-tune the model better to get efficient results.

B. Design and Implementation

1) MSCN:

a) Query encoding:

In order to use the queries from the standard IMDB database for our project, we need to encode the queries to feed them to the model. For any given query, there can be one or more tables, no join, one join or multiple joins, and one predicate or many predicates. For the featurization of the given SQL queries, we performed the encoding of every single query into sets of Joins, Tables, and Predicates in the same manner as described in the paper by Kipf et. al.

A query $q \in Q$ is represented as a collection $(T_{\mathrm{q}}, J_{\mathrm{q}}, P_{\mathrm{q}})$ of a set of tables $T_{\mathrm{q}} \subset T$, a set of joins $J_{\mathrm{q}} \subset J$ and a set of predicates $P_{\mathrm{q}} \subset P$ participating in the specific query q. T, J, and P describe the sets of all available tables, joins, and predicates, respectively. Each table $t \in T$ is represented by a unique one-hot vector v_{t} (a binary vector of length $|T|$ with a single non-zero entry, uniquely identifying a specific table). [4]

Similarly, joins $j \in J$ are featurised with a unique one-hot encoding. We performed encoding on all the predicates of the form (col, op, Val) using a categorical representation for columns col and operators op with respective unique one-hot vectors and represented Val as a normalized *value* $\in [0,1]$, using the Maximum and Minimum values of the attributes for each query. [4]

After the query representation ($\mathrm{T_q}$, $\mathrm{J_q}$, $\mathrm{P_q}$), our MSCN model takes the following form:

Table Module: $w_{\mathrm{T}} = \frac{1}{|T_{\mathrm{q}}|} \sum_{t \in T_{\mathrm{q}}} MLP_{\mathrm{T}}(v_{\mathrm{t}})$

Join Module: $w_{\mathrm{J}} = \frac{1}{|J_{\mathrm{q}}|} \sum_{j \in J_{\mathrm{q}}} MLP_{\mathrm{J}}(v_{\mathrm{j}})$

Predicate Module: $w_{\mathrm{P}} = \frac{1}{|P_{\mathrm{q}}|} \sum_{p \in P_{\mathrm{q}}} MLP_{\mathrm{P}}(v_{\mathrm{p}})$

Merge Predict: $w_{\mathrm{out}} = MLP_{\mathrm{out}}([w_{\mathrm{T}}, w_{\mathrm{J}}, w_{\mathrm{P}}])$ [4]

We performed the encoding for all the 21 tables and 108 columns present in the database. We decomposed the query clauses *Select, From,* and *Where* into different blocks to get more information about the tables, joins, and predicates. After gaining information about the attributes contributing in the joins, we a) Alphabetically sorted the attributes to perform concatenation later regardless of their position in a given query, b) Take the encoding of these attributes, and finally, c) Concatenated all the encoding.

b) Model Implementation:

The key ideas of model implementation are adopted from the paper by Kipf et. al. [4]. The model takes in a set of tables, joins and predicates as the input for a given set of queries. A fully connected neural network then transforms each input to an intermediate representation. These representations are then concatenated and fed to the layers which finally produce a predicted cardinality as output. We have used the following hyper-parameters for our experiments:

- Iterations: 5000 (No.of batches and passes formed)
- Batch size: 64 items (as fixed batch size, randomly sampled from the set of training queries)
- Activation function: such as- ReLU, Sigmoid function are used For modulating the output at the end of each layer
- Vector lengths: 256 (intermediate representation)
- Dropouts: A regularization technique, where we drop out the contribution of certain units randomly to the output of the overall network
- Layers: 2 layers for each set and concatenate the results, before passing it to the final layer

2) DeepDB:

To get the cardinality estimations, we create an ensemble of RSPN's for our IMDB database. This ensemble contains RSPN's for tables having a primary key and a foreign key relation. While creating the ensembles, we specify a budget factor (based upon the time and space) which acts as a constraint for the learning of RSPN's. The budget factor used in the ensemble selection is five and ten (i.e. the training of the larger RSPN's takes approximately five and ten times longer than the base ensemble respectively) while using 10^7 samples per RSPN. Each RSPN has to achieve a goal in which it should be attaining the maximum RDC values. which helps in optimizing the ensemble. [6] Along with these, we also use the variants which determine the granularity of clustering. In this way, we create an ensemble with various hyperparameters in place. Experimenting and tuning these values helped us achieve better results to evaluate against our other benchmark MSCN.

C. Research Questions

With our study, we aim to probe deeper into the following specific research problems:

- Using the MSCN model for Cardinality Estimation and studying the impact of changing the workload size on performance.
- Using the DeepDB model for Cardinality Estimation and studying the impact of changing the hyperparameters on performance.
- Comparison of the performance of these two models.

The subsequent sections describe the experimental setups and evaluation results for our listed research questions.

V. Experimental Setup

A. Dataset and Benchmark

For our research, we have selected the *IMDB dataset* which contains 22 tables with real-world information about movies, TV series, actors, directors, producers, etc. The dataset captures more than 2.5 M movie titles produced over 133 years by 234,997 different companies with over 4 M actors. The total size of the database is around 3.7 GB. The main reason for selecting this database is that the IMDB dataset consists of multiple correlations and non-uniform data distribution, and therefore proves to be very challenging for cardinality estimators. Since it has also been used for evaluation in previous research work, this facilitates the requirement of a real-world benchmark database.

B. JOB, JOB-Light, Synthetic Queries

For our experiments and evaluation we used three different query workloads: a) a synthetic workload generated with 5,000 unique queries containing zero to two joins, b) another synthetic workload scale with 500 queries to showcase model generalization to more joins, and c) JOB-light, a workload derived from the Join Order Benchmark (JOB) containing 70 of the original 113 queries (does not contain any predicates on strings nor disjunctions and only contains queries with one to four joins).

Exp. Setup	No. of Joins	Budget Factor	Max. Variants
1	3	5	1
2	6	10	1
3	6	5	5
4	6	5	1
5	5	5	5
6	4	5	1

TABLE I: Different experimental setups for the DeepDB Model

Experimental Setup	MSE	MAPE
1	1.531	0.290
2	1.328	0.425
3	4.710	0.539
4	2.935	0.309
5	1.290	0.665
6	2.857	0.329

TABLE II: Error values for different setups in DeepDB

For performing the experiments and obtaining results we configured our environment with the following settings:

- Operating System : Ubuntu 18.04
- GPU : NVIDIA 1050 GTX
- CPU : Intel i5 8th generation
- Device memory : 8 GB

With this configurations we execute our models for both MSCN [4] and DeepDB. [6]

VI. Evaluation

In this section, we discuss the experiments we performed to obtain the results and then analyze them. As discussed in the previous sections, we are using DeepDB and MSCN as our major execution platform on the IMDb database. For evaluating the models, the queries which we have used consist of joins ranging from 1 to 4 and are derived from the JOB queries.

A. Experiment 1: Trained and experimented with the Deep DB model on various hyperparameter values like number of joins, budget factor and max variants

For this experiment, six different configurations with different parameter settings were selected for deducing the robustness of the model as described in Table I. The first setup is the default setting which gave us the optimal results to compare the results of other experiments with.

Table II shows the error values for the different configuration setups in DeepDB. The plot for MSE in Fig 8 shows the distribution of errors over all 70 queries, distinguishing them based on the number of joins present in the query. Similarly for MAPE, the plot in Fig 9 shows the error distribution for all the different queries.

After plotting and comparing the errors for various experimental hyperparameters setups in DeepDB, we observe that our experimental setups 3 and 5 showed us the worst results compared to all the experiments we

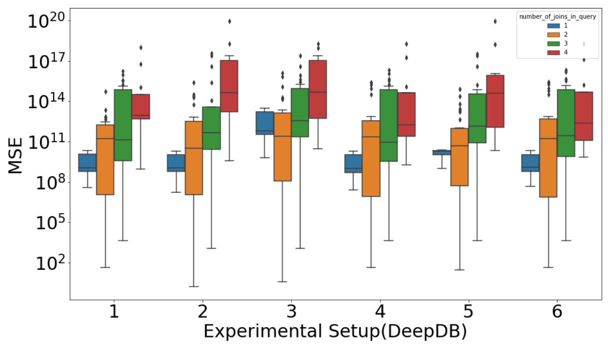

Fig. 8: Mean Squared Error for various experimental setup over number of joins performed using DeepDB, (y-scale as log scale).

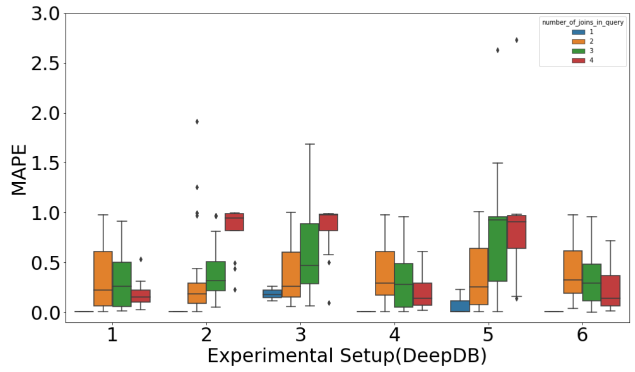

Fig. 9: Mean Absolute Percentage Error for DeepDB

performed. Amongst both these setups, we had common values for the number of joins and max variants. Experimental setups 1, 4, and 6 showed the best results with the least errors and good distributions of the error rates as compared to the other experiments conducted.

B. *Experiment 2: MSCN hyperparameters trained on synthetic queries and tested on JOB-light queries*

For this experiment, Table III shows the details for all the configurations used. We have applied different dropout configurations in our model. The dropout values in the table signify the percentage of nodes to be dropped in a given layer of the network. In our case, we have used all the layers of the network as mentioned in the Layers column of the table.

Exp. Setup	Configurations	Dropout Rate	Layers
1	Steer = False	0.80	All Layers
2	Steer = True	0.85	All Layers
3	Steer = False	0.90	All Layers
4	Steer = True	0.70	All Layers
5	Steer = False	1	All Layers
6	Steer = True	1	All Layers

TABLE III: Different dropout rates applied on the test dataset

Experimental Setup	MSE	MAPE
1	2.168	3.709
2	2.246	4.089
3	2.157	24.11
4	1.938	182.169
5	2.194	13.331
6	2.232	2.895

TABLE IV: Error values for different setups in MSCN

Regressors	MSE	RMSE
Decision Tree Regressor	0.0625	0.2500
MLP Regressor	0.0477	0.2185
Support Vector Regressor	0.0621	0.2492
Random Forest Regressor	0.0509	0.2256
Linear Regressor	0.0775	0.2785
Lasso Regressor	0.0505	0.2247
Ridge Regressor	0.0776	0.2787

TABLE V: MSE and RMSE for Regressors in MSCN

We used MAP and MSE to see how the error rate differs for all the six different configurations on the test dataset. Table IV shows the error values for the different setups used in our experiment. From the table, comparing setups 5 and 6, we can conclude that the MAPE increases if we don't steer and keep the dropout configuration= 1.0 i.e. no dropout at all in the model. On the other hand, if we steer with less dropout rate then the MAPE shows an abrupt rise in the error rate for the overall dataset as observed for setup 4.

We also plotted the error rates MAP and MSE for the different number of joins presented in the queries which are depicted in Fig 10 and 11 respectively. From Fig.8, it is observed from a comparison between the MAPE for setups 2 and 4 that steering on that dropout configuration reduces the variations on more than 2 joins, but is not too beneficial on queries with a single join. Whereas, on MSCN's MSE, for all of the setups we are getting similar variation based on the number of joins present in the queries.

C. *Experiment 3: MSCN Regressors trained on synthetic queries and tested on JOB-light queries*

In this experiment, we examine how well our model competes with the other ML models presented with the same data. We have performed this experiment with seven different regressors and computed the MSE and RMSE on the test data i.e. JOB-light queries.

Table V shows both the error vslues for the various regressors settings. From the table, we observe that the MLP Regressor is the one with the best MSE and RMSE followed by Support Vector Regressor and Random Forest Regressor respectively. If we make a comparison with our model results, then our model is showing better MSE loss against the MSE of MLP Regressor.

From the evaluated results, it is evident that the MSCN model performs better for tables containing multiple joins(2, 3, and 4) as compared to the tables con-

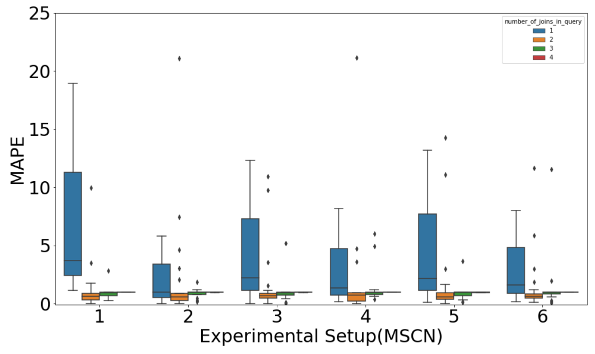

Fig. 10: Mean Absolute Percentage Error for MSCN

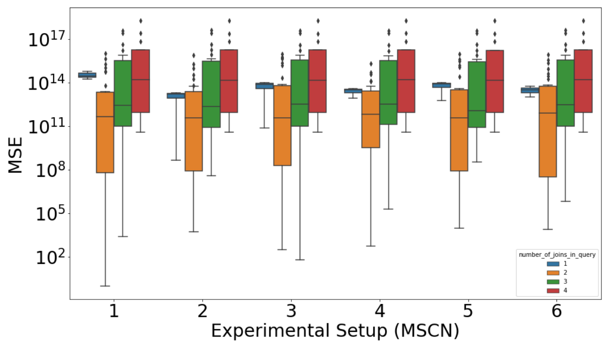

Fig. 11: Mean Squared Error for various experimental setup over number of joins performed using MSCN, (y-scale as log scale)

taining a single join. MSCN captures the correlations effectively in tables. For tables containing single joins, the DeepDB model performs better than the MSCN with smaller MAPE distributions. This might be reasoned with the architecture of an RSPN to effectively translate the primary key- foreign key relationship in a table. The DeepDB model also shows an overall good performance in the case of multiple joins with low error rates.

With this, we conclude our evaluation. In the next section, we summarize our work with suggestions for future work and refinements.

VII. Conclusion & Future Work

With the need for optimizing query performance, it becomes crucial to develop new strategies for performing effective cardinality estimation. The existing traditional techniques in relational data management may not be suitable for tables having join-crossing correlations. In this paper, we presented the concepts replicating the work of Kipf et. al. on Multi-set Convolutional Networks (MSCNs) and the work of Hilprecht et.al on DeepDbs. We provide a comparative study of the models on the IMDB dataset using the benchmark JOB queries. We estimate cardinalities using both workload-driven MSCN and data-driven DeepDB to find out which among these two models give a better cardinality estimation and is optimal for real-world data. We describe the different parameter settings in which we performed the experiments to arrive at a conclusive model.

Our framework is evaluated based on Mean Squared Error (MSE) and Mean Absolute Percentage Error(MAPE). Although MSCNs are robust in the case of multiple correlations being present in the tables, the use of RSPN's in DeepDB helps in training data in optimal time thereby providing good estimates for all the possible queries over the database. It was challenging to encode the queries in a manner to be effectively utilized in the MSCN framework. Since MSCN uses encoded queries and DeepDB takes non encoded data as input, DeepDB is found to have a higher model interpretability among the two.

This paper is open to future improvements and refinements such as evaluating an ensemble of MSCN and DeepDB and reviewing their performance. Since the representation of the join components in MSCN is quite naive, it would be appealing to add different representations for MSCNs like graph representations. The approaches also have a few limitations according to their applications and need further research to find a solution. Most of them cannot support string predicates for prefix matches. Furthermore, these solutions would need to adapt to the use of nested/sub-queries and special features (e.g. JSON support in the SQL standard). Similarly, their integration with other system components can also be studied - for e.g how exactly the query optimizer uses the information from these models or how the learned representations of MSCN can be used for other aspects (e.g for reinforcement learning acceleration of the search-space exploration process).

References

[1] "Cardinality estimation for performance tuning," https://www.perftuning.com/blog/sql-server-2014-cardinality-estimator/.

[2] P. Mishra and M. H. Eich, "Join processing in relational databases," *ACM Computing Surveys (CSUR)*, vol. 24, no. 1, pp. 63–113, 1992.

[3] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "Query optimization through the looking glass, and what we found running the join order benchmark," *The VLDB Journal*, vol. 27, no. 5, pp. 643–668, 2018.

[4] A. Kipf, T. Kipf, B. Radke, V. Leis, P. Boncz, and A. Kemper, "Learned cardinalities: Estimating correlated joins with deep learning," *arXiv preprint arXiv:1809.00677*, 2018.

[5] X. Zhou, C. Chai, G. Li, and J. Sun, "Database meets artificial intelligence: A survey," *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[6] B. Hilprecht, A. Schmidt, M. Kulessa, A. Molina, K. Kersting, and C. Binnig, "Deepdb: learn from data, not from queries!" *arXiv preprint arXiv:1909.00607*, 2019.

[7] M. Zaheer, S. Kottur, S. Ravanbakhsh, B. Poczos, R. R. Salakhutdinov, and A. J. Smola, "Deep sets," in *Advances in neural information processing systems*, 2017, pp. 3391–3401.

[8] H. Poon and P. Domingos, "Sum-product networks: A new deep architecture," in *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*. IEEE, 2011, pp. 689–690.

[9] A. Molina, A. Vergari, N. Di Mauro, S. Natarajan, F. Esposito, and K. Kersting, "Mixed sum-product networks: A deep architecture for hybrid domains," in *Thirty-second AAAI conference on artificial intelligence*, 2018.