A Hybrid Cost Model for Evaluating Query Execution Plans

Ning Wang

Thesis submitted to the University of Ottawa
in partial Fulfillment of the requirements for the
Master of Computer Science (Applied Artificial Intelligence)
in Electrical and Computer Engineering

School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

Abstract

Query optimization aims to select a query execution plan among all query paths for a given query. The query optimization of traditional relational database management systems (RDBMSs) relies on estimating the cost of the alternative query plans in the query plan search space provided by a cost model. The classic cost model (CCM) may lead the optimizer to choose query plans with poor execution time due to inaccurate cardinality estimations and simplifying assumptions [3,4,5]. A learned cost model (LCM) based on machine learning does not rely on such estimations and learns the cost from runtime [23,29,48]. While learned cost models are shown to improve the average performance, they may not guarantee that optimal performance will be consistently achieved. In addition, the query plans generated using the LCM may not necessarily outperform the query plans generated with the CCM. This thesis proposes a hybrid approach to solve this problem by striking a balance between the LCM and the CCM. The hybrid model uses the LCM when it is expected to be reliable in selecting a good plan and falls back to the CCM otherwise. The evaluation results of the hybrid model demonstrate promising performance, indicating potential for successful use in future applications.

Acknowledgements

I would like to express my sincere gratitude to my advisor, Prof. Verena Kantere, for her constant support and encouragement during the difficult years of the COVID-19 pandemic, which enabled me to get through the toughest moments of my life. Her profound knowledge in the database field, her rigor and passion for research, and her patience have benefited me greatly in this journey towards my master's degree. Without her guidance, I would not have been able to complete my degree.

I would also like to thank Amin Kamali, a PhD candidate at the University of Ottawa, for his deep background in machine learning and constructive advice that has always helped me solve my work problems, and for taking care of my studies and life along the way.

As well, I would like to thank IEEE AIKE, CASCON 2022 and the team for the IBM CAS project for their approval and advice on my research.

Last but not least, I would like to thank my thesis examiners, Prof. Iluju Kiring and Prof. Tet Yeap from the University of Ottawa, for their valuable and detailed comments and all the time and effort they have invested in the evaluation of my thesis.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

RDBMSs	Relational Database Management Systems
CCM	Classic Cost Model
LCM	Learned Cost Model
SQL	Structured Query Language
TPC-DS	Transaction Processing Performance Council Decision Support Benchmark
MCMC	Markov Chain Monte Carlo
CNNs	Convolutional Neural Networks
RNNs	Recurrent Neural Networks
ACO	Ant Colony Optimization
RL	Reinforcement Learning
MDP	Markov Decision Process
LSTM	Long Short-Term Memory
GRU	Gated Recurrent Unit
BLEU	Bilingual Evaluation Understudy
ROUGE	Recall-Oriented Understudy for Gisting Evaluation
UDF	User-Defined Functions
BP	Back-Propagation
MSCN	Multi-set Convolutional Network
MLPs	Multilayer Perceptrons
TCNN	Tree Convolutional Neural Network

Chapter 1

Introduction

1.1 Introduction of Query Optimization

1.1.1 Query Optimization and Query Plan

Querying is a manifestation of human needs, from looking up words in dictionaries to finding bargains on Amazon. Queries appear in our daily life all the time. In the face of massive needs for querying, if we can effectively reduce query execution time, it will undoubtedly improve the efficiency of our work and life. In relational databases, queries are written using SQL statements. A SQL query tells the relational database "what to ask", but it is up to the database to decide “how to ask” for it. Generally, the database transforms the query into a number of query plans. Each plan can be represented as a tree, with each node corresponding to an operator (table scan, merge join, etc.) that works bottom-up, from the leaf nodes to the root node, to get the query results. Figure 1[25] shows an example query that joins two tables in the TPC-DS dataset, along with two alternative query execution plans. Although different query plans for the same SQL query can yield the same output, the time and resources (e.g., CPU, memory) required for executing the query vary greatly. Given a query, the optimal query plan may obtain the query result in sub-seconds, while a bad query plan may take several hours to run. Therefore, choosing a good query plan can help save a tremendous amount of time and thus improve productivity.

```
SELECT
  TPCDS.CUSTOMER_ADDRESS.CA_ADDRESS_SK2,
  TPCDS.STORE_SALES.SS_PROMO_SK
FROM
  TPCDS.CUSTOMER_ADDRESS
  INNER JOIN TPCDS.STORE_SALES
  ON TPCDS.STORE_SALES.SS_ADDR_SK =
TPCDS.CUSTOMER_ADDRESS.CA_ADDRESS_SK
WHERE
  +00110.90 = TPCDS.STORE_SALES.SS_COUPON_AMT AND
  +00206.50 <= TPCDS.STORE_SALES.SS_NET_PAID;
```

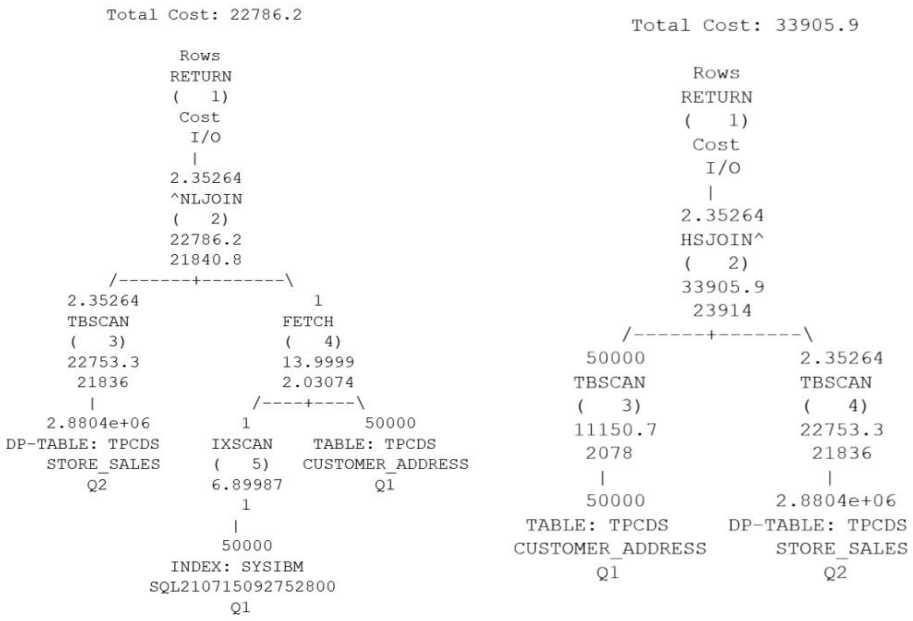


Figure 1: A query for two tables from TPC-DS and two alternative query plans for executing the query [1]

1.1.2 Query Optimizer and Cost Model

A traditional RDBMS has a module, the "query optimizer", which is dedicated to select an optimal query plan from the search space. The query optimizer compares alternative plans in the search space using a "cost model" that uses statistics from the underlying data, as well as environmental specifications such as hardware and concurrency settings. The cost model estimates the cost of each operator and accumulates the costs of all operators in the plan to

estimate the total cost. The query optimizer uses the cost model to evaluate different query plans in its search space in order to choose an optimal plan with the lowest total cost.

The "cost model" mentioned above that the query optimizer relies on is a traditional or classic cost model (CCM). With the development of machine learning, a new learning-based cost model has emerged.

Unlike the classic cost models(CCM) that utilizes heuristic-based approaches and relies on statistical information about the data (e.g., the number of rows in a table, the distribution of key values, the availability of indexes, etc.) to estimate the cost, the learned cost model uses various machine learning techniques such as regression, decision trees, neural networks or deep learning to learn patterns in query execution, learns cost from a large amount of historical query data, such as actual execution time.

1.1.3 Existing Challenges in Cost Model

In the process of query optimization, the statistical information (cardinality estimation) that CCM relies on is not always accurate, and even has a large deviation from the actual value. Furthermore, the queries do not always satisfy the assumptions. All these may lead the optimizer to choose query plans with poor execution time.

A learned cost model (LCM) does not rely on such statistics and learns the cost from runtime. While LCMs improve average performance, they may not guarantee that optimal performance is consistently achieved. Besides, the query plans selected using the LCM may not necessarily outperform the query plans generated with the CCM. In section 1.4 of this chapter, we will use an example to illustrate.

1.2 Motivation and Problem Statement

Both traditional cost models and learned cost models suffer from the problem of inaccurate predictions that originates in either cardinality misestimations, simplifying assumptions, or lack of appropriate and adequate training data. Although learned cost models (LCM)s are able to improve the average performance compared to the traditional (classic) cost model (CCM), we found that the LCM does not always outperform CCM through experimentation. To compare the performance of LCM and CCM, we compute the Suboptimality of LCM compared to CCM, as:

$$\mathrm{SubOpt}_{q_i}(\mathrm{LCM}, \mathrm{CCM}) = -\log_{10}\frac{\mathrm{ET}_{\mathrm{LCM}}(q_i)}{\mathrm{ET}_{\mathrm{LCM}}(q_i)} \quad (1)$$

where represents the execution time of the query using approach. With this formulation, values greater than zero represent an improvement, while the ones less than zero represent regression.

Figure 2 shows the distribution of the for a set of test queries. The positive (blue) bins represent the scenarios where the LCM outperforms the CCM, and the negative (red) bins represent the scenarios where the CCM outperforms the LCM. Although the LCM outperforms the CCM in most cases, the number of cases where it regresses is still significant and cannot be ignored. This means that in a realistic setting, there would be a significant risk in substituting the CCM with the LCM, as the latter would perform poorly for many queries in a workload. Thus, the LCM cannot completely replace the CCM.

In addition to the aforementioned considerations, there are additional aspects that require attention when utilizing LCMs. These factors are closely related to the previously mentioned points.

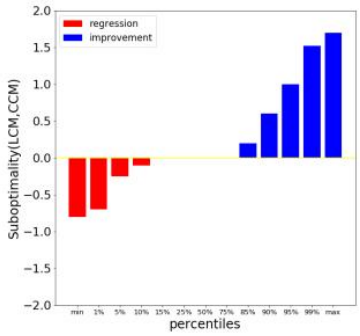


Figure 2: Distribution of SubOpt (LCM, CCM) for the test set

When an LCM is appropriately trained, its performance is expected to be satisfactory, provided that: a) The queries in the workload and the plans being evaluated closely resemble the examples present in the model's training data, b) The underlying data distribution has not undergone significant changes since the model was trained, and c) The environment variables and settings during evaluation are consistent with those employed during the model's training.

While addressing these concerns is crucial for maximizing the effectiveness and reliability of LCMs, it is important to first develop a technique that can leverage the optimal performance of both the CCM and the LCMs. By creating such a technique, we can later adapt it to address the aforementioned concerns as well. Consequently, our primary objective is to answer the following question: How can we conceptualize a technique that consistently selects a suitable query plan, utilizing both the CCM and the LCMs?

1.3 Contribution

In this work, we propose a hybrid cost model as a technique that can leverage the best performance of a CCM and a set of LCMs. The hybrid cost model can strike a balance between the LCM and the CCM. To do so, a set of LCMs are trained each specialized on a certain class of

queries. These LCMs together with the CCM, make up a collection of 'base models'. Then alternative plans for each query in the training set are evaluated using each of the base models. The quality of the estimates generated by each model is determined by evaluating the correlation between the estimates and runtime. Each query is then labeled by the base model that produces estimates with the highest correlations. Then a classification model is trained to take queries as input and predict which base model(s) would produce the best plan.

In summary, this thesis makes the following contributions:

1. It proposes a novel architecture that benefits from the advantage of using LCMs while minimizing regressions by falling back on the CCM when necessary.
2. It proposes a query classifier that learns to route queries to a base cost model (either learned or classic) that is expected to provide estimates with a higher correlation with the runtime.

Besides, the thesis was published in IEEE AIKE [49] and IBM CASCONxEVOKE [50], and the Patent has been filed with the Patent Office in United States of America [51].

1.4 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 introduces the background of query optimization and related literature review. Chapter 3 presents the proposed approach for building the hybrid cost model. Chapter 4 describes the experimental setup used for evaluating the proposed method. Chapter 5 outlines the evaluation results. Finally, Chapter 6 provides conclusions and directions for future works.

Chapter 2

Background and Literature Review

2.1 Background

Query optimization involves several fields and directions. In recent years, with the development of machine learning techniques, many machine learning techniques have been applied to query optimization. In this chapter, we will introduce three important directions of query optimization and the application of machine learning in that direction.

2.1.1 Cardinality Estimation

The query optimizer relies on cost model to obtain the optimal query plan. And cost model relies on the statistics of the underlying data, in particular, the size of the data flowing through each operator, also known as the "cardinality". However, the true cardinalities are typically unknown at compile-time. Therefore, the optimizer uses various methods to estimate them.

Traditional methods in Cardinality Estimation

Histogram**s:** A histogram is a statistical representation of the data distribution in a column. It divides the column's values into buckets or ranges and estimates the number of values falling into each bucket. This information can be used to estimate the cardinality.

Sampling: Sampling involves randomly selecting a subset of the data from a column or table and analyzing it to estimate the cardinality [2]. The cardinality of the sample is then extrapolated to estimate the cardinality of the entire dataset.

Statistical models: Some DBMSs use statistical models, such as the Bayesian model or the Markov Chain Monte Carlo (MCMC) method, to estimate cardinality based on observed data patterns.

Metadata and heuristics: DBMSs often utilize metadata, such as index statistics or data distribution information, along with heuristics to estimate cardinality. These estimates may be based on assumptions about the data or historical query execution patterns.

The purpose of cardinality estimation is to predict the number of rows that a query is likely to process through each plan operator without executing the query plan. The query optimizer uses the result of the cardinality estimate to compute the total cost of the alternative plans and ultimately select the best one, i.e., the plan with the lowest cost. However, cardinality estimation is not always accurate, because realistic databases hardly satisfy the assumptions of independence and uniformity [3,4,5] which are typically used in classic estimation methods.

Machine Learning methods applied to Cardinality Estimation

The inaccuracy in the cardinality estimates and the simplifying assumptions used in cost models have motived an outpouring of research in the area with Machine Learning.

(1) Clustering

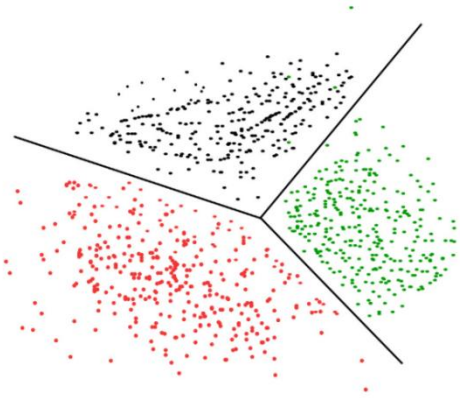


Figure 3: An example of Clustering [6]

As shown in Figure 3, clustering is a technique used in machine learning and data analysis to group similar data points together based on their characteristics or attributes [7]. It is an unsupervised learning method, meaning it does not require labeled data or predefined classes. The goal of clustering is to find inherent patterns or structures within a dataset by organizing the data points into clusters, where points within the same cluster are more similar to each other compared to those in different clusters. The similarity or dissimilarity between data points is typically measured using a distance metric, such as Euclidean distance or cosine similarity. Clustering algorithms aim to minimize the intra-cluster distance (distance between points within the same cluster) while maximizing the inter-cluster distance (distance between points in different clusters). Clustering techniques can be applied in cardinality estimation to help analyze and understand the distribution of data in a dataset. Here are a few ways clustering can be utilized for cardinality estimation:

Data Exploration: Clustering can be used to explore the structure of the data. By grouping similar data points together, it becomes easier to identify patterns, outliers, and potential clusters that represent distinct entities. This analysis helps in understanding the cardinality of different groups within the dataset.

Feature Engineering: Clustering can also be employed as a feature engineering technique to create new features that capture the similarity between data points. These features can then be used in cardinality estimation models to improve their accuracy. For example, clustering-based distance metrics, such as the distance to the centroid of a cluster, can be used as features to estimate the cardinality of a specific entity.

Sampling: Clustering can aid in selecting representative samples from a large dataset for cardinality estimation. Instead of analyzing the entire dataset, clustering allows for the selection of a subset of clusters that can adequately represent the entire dataset. This approach can

significantly reduce the computational cost of cardinality estimation while still providing reliable estimates.

Anomaly Detection: Clustering algorithms can help identify anomalies or outliers in the data, which can impact the accuracy of cardinality estimation. By detecting and treating outliers separately, more accurate estimates can be obtained for the remaining data points.

Data Preprocessing: Clustering can be utilized as a preprocessing step before cardinality estimation. It can be applied to remove redundant or highly correlated attributes from the dataset, reducing the dimensionality and improving the accuracy of the cardinality estimation process.

(2) Deep learning

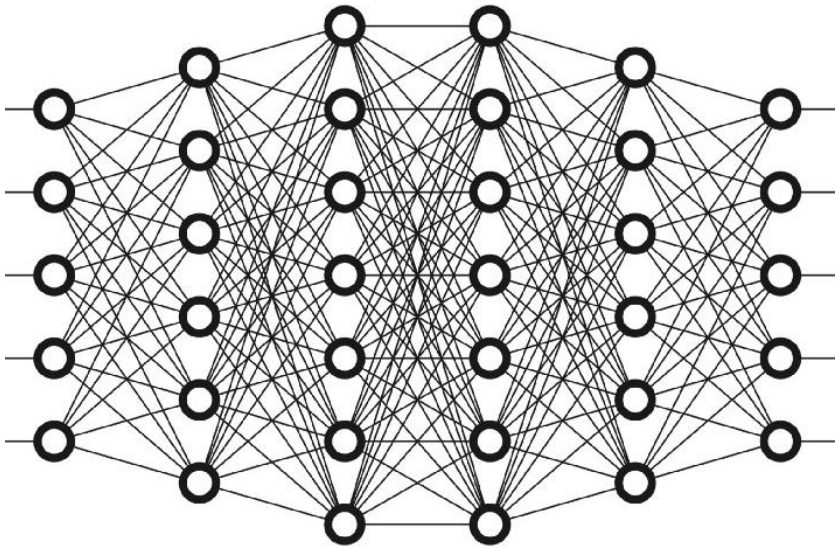


Figure 4: The architecture of Artificial Neural Networks [8]

Deep learning is a subfield of machine learning that focuses on artificial neural networks [8]. It aims to enable computers to learn and make intelligent decisions by automatically extracting meaningful patterns and representations from large amounts of data. Deep learning models are constructed with multiple layers of interconnected artificial neurons called artificial neural

networks (shown in Figure 4). These networks are organized into an input layer, one or more hidden layers, and an output layer [9]. Each neuron takes in a set of inputs, applies a mathematical operation to them, and produces an output that is passed to the neurons in the next layer. This process is repeated through the network until the final output is generated. Here are some ways deep learning can be utilized for cardinality estimation:

Feature Learning: Deep learning models can learn relevant features from data. In the context of cardinality estimation, this means that neural networks can learn representations of the data that capture important patterns and characteristics related to distinct values. By extracting meaningful features, deep learning models can enhance the accuracy of cardinality estimations.

Neural Network Architectures: Deep learning allows for the design of complex neural network architectures that can capture intricate relationships within the data. Various types of neural networks, such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs) [8], can be used to model the structure of the data and improve cardinality estimation performance.

Supervised Learning: Deep learning models can be trained using supervised learning techniques, where they are provided with labeled data consisting of input features and corresponding cardinality values. By leveraging labeled datasets, neural networks can learn from the examples and generalize to make accurate cardinality estimations on unseen data.

Ensemble Methods: Deep learning models can be combined with traditional cardinality estimation techniques through ensemble methods. For example, an ensemble of deep learning models and statistical estimators can be used to obtain accurate and robust cardinality estimations. The deep learning models can capture complex patterns, while the statistical estimators can provide a baseline or refine the results further.

Transfer Learning: Deep learning models trained on one dataset can be fine-tuned or transferred to a different dataset for cardinality estimation. Transfer learning allows leveraging the learned

representations and knowledge from one domain to another, even when the datasets differ. This can be particularly useful when labeled data for the target dataset is scarce or unavailable.

2.1.2 Join Order Enumeration

Join order enumeration is a technique used in query optimization for relational databases. When executing a query involving multiple tables, the database optimizer needs to determine the order in which the tables are joined together. The join order can significantly impact the performance of the query, as different join orders may result in different intermediate results and execution plans. Join order enumeration involves considering all possible permutations of table join orders and evaluating the cost of each permutation. The cost is typically measured based on factors like the number of disk accesses, CPU usage, and network communication required for the join operation. The goal of join order enumeration is to find the join order with the lowest estimated cost. Enumerating all possible join orders is a combinatorial problem, as the number of possible permutations grows exponentially with the number of tables involved. For example, if a query involves three tables, there are six possible join orders (3! = 3 x 2 x 1). As the number of tables increases, the number of possible permutations grows rapidly, making exhaustive enumeration impractical for queries with many tables.

Machine Learning methods applied to Join Order Enumeration

To address the issue above, query optimizers often employ machine learning methods to reduce the search space and find a reasonably good join order without considering all permutations.

(1) Ant Colony Optimization

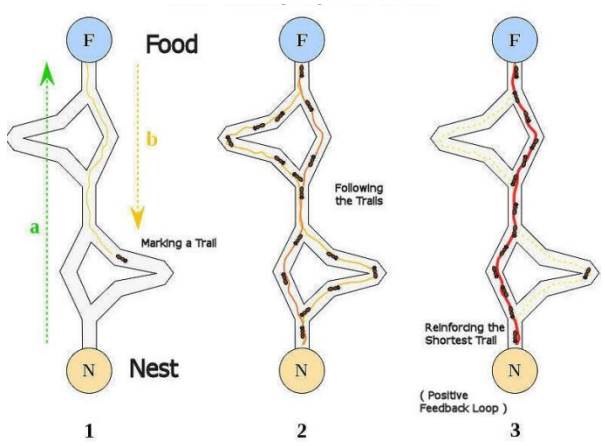


Figure 5: The conception of Ant Colony Optimization [10]

Ant Colony Optimization (ACO) is a metaheuristic algorithm inspired by the behavior of ants searching for food [10]. It is used to solve optimization problems, particularly combinatorial optimization problems. As shown in Figure 5, ACO algorithms are based on the observation that ants can collectively find the shortest path between their colony and a food source by depositing and following pheromone trails [11].

In the context of ACO, a problem is represented as a graph, where nodes represent problem-specific elements, such as cities, and edges represent connections or paths between them[11]. The goal is to find the optimal path or combination of elements based on a defined objective function.

The ACO algorithm consists of a population of virtual ants that iteratively build solutions by moving through the graph. Initially, each ant is placed on a random node, and at each step, it chooses the next node to move to base on a probabilistic decision rule, often referred to as the "transition rule."

The decision of each ant is influenced by two main factors: the amount of pheromone on the edges and the heuristic information, which represents the desirability of choosing a particular path based on problem-specific knowledge. As ants move, they deposit pheromone on the edges they traverse, and the amount of pheromone is updated based on the quality of the solutions found.

Over time, ants tend to follow the paths with higher pheromone levels, as these paths become more attractive due to positive feedback. By iteratively repeating the ant movement and updating the pheromone levels, the algorithm converges towards an optimal or near-optimal solution.

(2) Reinforcement Learning

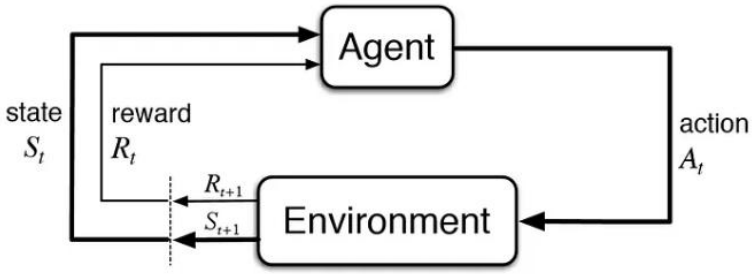


Figure 6: The architecture of Reinforcement Learning [12]

Reinforcement learning (RL) is a type of machine learning that involves an agent learning to make decisions in an environment to maximize its cumulative reward (shown in Figure 6). It is inspired by how humans and animals learn from the consequences of their actions [12].

In reinforcement learning, an agent interacts with an environment, receives feedback in the form of rewards or penalties, and learns to take actions that lead to the highest possible reward [12]. The agent learns through a trial-and-error process, exploring different actions and observing the outcomes to understand which actions are more favorable.

The environment is typically represented as a Markov Decision Process (MDP), which consists of states, actions, transition probabilities, and rewards. At each step, the agent observes the current state, selects an action, and the environment transitions to a new state based on the action taken [11]. The agent receives a reward or penalty based on the state transition, providing feedback to guide its learning. This process is repeated until the agent reaches a terminal state [11].

Here's a high-level overview of how RL can be applied to join order enumeration:

State Representation: The first step is to define the state representation. The state typically includes information about the query, such as the tables, their sizes, cardinalities, and join predicates.

Action Space: Define the action space, which represents the possible actions the RL agent can take. In join order enumeration, each action corresponds to selecting the next table to join.

Rewards: Define the reward function that provides feedback to the RL agent. The reward can be based on query execution time, resource utilization, or any other performance metric. The goal is to maximize the reward over time [11].

Training: Train the RL agent using techniques like Q-learning or policy gradient methods. The agent interacts with the environment by selecting actions (choosing the next table to join) based on its current state [12]. The agent receives rewards based on the performance of the chosen join order and updates its policy to maximize future rewards.

Exploration-Exploitation Tradeoff: Balancing exploration and exploitation is crucial in RL. Initially, the agent may explore different join orders to learn their performance. As training progresses, the agent can exploit its learned policy to choose join orders with higher expected rewards.

2.1.3 Query Rewriting

Query rewriting refers to the process of transforming or modifying a given database query into an equivalent query that can be executed more efficiently or effectively. It involves manipulating the structure or content of the original query while preserving its intended meaning and semantics. Query rewriting is often employed in database optimization and query processing to improve query performance, enhance result accuracy, or adapt the query to a different data model or system.

Machine Learning methods applied to Query rewriting

(1) Recurrent Neural Network

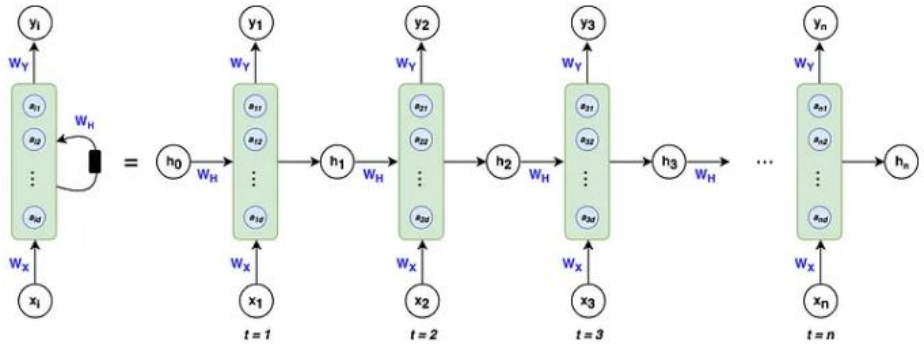


Figure 7: The architecture of Recurrent Neural Network[13]

A Recurrent Neural Network (RNN) is a type of artificial neural network that is specifically designed to process sequential data [13].

As shown in Figure 7, the key characteristic of an RNN is its ability to maintain an internal state or memory that allows it to process inputs in a sequential manner. This memory allows the network to retain information about previous inputs and use it to influence the processing of future inputs [14]. In other words, an RNN has a form of "recurrence" that allows it to learn patterns and relationships in sequential data.

The basic building block of an RNN is the "recurrent layer," which consists of a set of interconnected nodes, often referred to as "memory cells" or "hidden units." Each node in the recurrent layer receives input not only from the current time step but also from its own output in the previous time step [15]. This feedback loop allows the network to maintain information about past inputs.

One popular variant of the RNN is the Long Short-Term Memory (LSTM) network [16], which was designed to address the "vanishing gradient" problem that can occur during training[16]. The

LSTM introduces additional mechanisms, such as input and forget gates, to control the information through the network and make the network to capture long-term dependencies in the data easily.

Here's a general outline of how RNNs can be applied to query rewriting:

Data Preparation: The first step is to gather a dataset of original queries and their rewritten versions, which serve as training examples. This dataset needs to be properly annotated, indicating the correct rewritten form for each original query.

Sequence Encoding: Each query is represented as a sequence of tokens, such as words or subwords. These tokens are then encoded into a numerical representation, typically using techniques like word embeddings or subword embeddings (e.g., Word2Vec, GloVe, or FastText).

Recurrent Neural Network Architecture: A common choice for sequence-to-sequence tasks like query rewriting is the Long Short-Term Memory (LSTM) [17] or Gated Recurrent Unit (GRU) networks [18]. These RNN architectures are designed to capture long-term dependencies and handle variable-length input sequences.

Encoder-Decoder Setup: The RNN is trained in an encoder-decoder framework. The encoder processes the original query tokens, generating a fixed-length vector representation known as the "context vector" or "thought vector." The decoder takes this context vector and generates the rewritten query tokens one by one.

Training: The RNN is trained using pairs of original queries and their corresponding rewritten versions. During training, the model learns to minimize the difference between its generated rewritten queries and the ground truth rewritten queries. This is typically done by minimizing a loss function like cross-entropy loss.

Inference: Once the RNN is trained, it can be used to rewrite new, unseen queries. Given an original query, the encoder processes it to obtain the context vector. The decoder then generates

the rewritten query by sampling tokens based on the context vector and its own internal state. This process continues until an end token or a maximum length is reached.

Evaluation: The rewritten queries generated by the RNN can be evaluated using metrics such as BLEU (Bilingual Evaluation Understudy) [19]. These metrics compare the generated rewritten queries against the ground truth rewritten queries to measure their similarity.

(2) Transformer

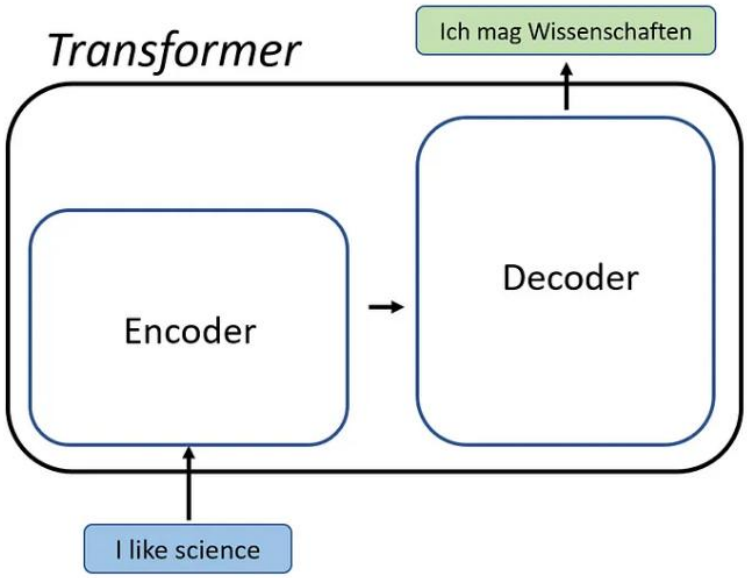


Figure 8: The architecture of Transformer [20]

A transformer is a type of neural network architecture that was introduced in a 2017 paper titled "Attention is All You Need" by Vaswani et al [21].

The key idea behind the transformer is self-attention, which allows the model to weigh the importance of different words or tokens in a sequence when processing it. Instead of relying on recurrent connections or fixed-length convolutions, the transformer processes the entire input sequence simultaneously.

As shown in Figure 8, the architecture of a transformer consists of an encoder and a decoder. The encoder takes an input sequence and processes it to create a representation of the input. The decoder then takes the encoder's representation and generates an output sequence.

The self-attention mechanism in a transformer enables the model to capture the dependencies between different words or tokens in a sequence more effectively than previous approaches. It allows the model to attend to different parts of the input sequence based on their relevance to each other, capturing long-range dependencies and improving the modeling of context [22].

Here's how the Transformer can be applied to query rewriting:

Encoding the original query: The original query is first tokenized into a sequence of tokens. Each token is then embedded into a dense vector representation, which captures both its semantic and positional information. The Transformer's encoder takes these embeddings as input and generates contextualized representations of each token in the query.

Generating alternative queries: Once the original query is encoded, different rewriting strategies can be employed to generate alternative queries. These strategies can involve paraphrasing, synonym replacement, expansion, or any other technique aimed at modifying the original query. The Transformer can be used to generate alternative queries by conditioning the decoder on the encoded representation of the original query.

Attention mechanism: The Transformer's attention mechanism plays a crucial role in query rewriting. It allows the model to attend to different parts of the input query during the rewriting process. By attending to relevant tokens, the Transformer can learn to modify specific aspects of the query, ensuring that the rewritten query maintains the user's intent while addressing any issues or limitations of the original query.

Because query optimization covers a wide range of domains and directions, we focus the attention on the cost model on which the query optimizer depends. So in the literature review, we focus more on the work that is related to the cost model.

2.2 Literature Review

The traditional (classic) cost model (CCM) is often criticized for its inaccuracy. Cardinality estimation is the "Achilles' heel"[23]. Traditional cardinality estimation methods can be divided into two main categories: (1) statistical histogram-based estimation and (2) sampling-based estimation. However, both methods have their own drawbacks. Statistical histogram-based estimation is subject to the assumptions of independence and uniformity, resulting in estimates that are often lower than the true values, and obtaining the true distribution of the data is also a challenge for this method. Sampling-based distributions, on the other hand, are constrained by the number of samples taken, and in addition to that, the accuracy of estimation drops dramatically when multiple table joins are involved. And these problems have been studied for more than thirty years and still have not been completely solved. So, for a long time, building a good query optimizer was "an art that only a few experts could fully master"[23], and even then, those few experts needed a lot of engineering time to carefully tune it to improve the query performance of a particular database. On top of that, they required tedious maintenance. As a result, commercial optimizers have held a firm grip on the market, and no free open-source query optimizer can match them. However, even for commercial optimizers, the problem of inaccurate cardinality estimation still exists, resulting in poor query performance [24].

However, over time, Lakshmi et al. [25] found that machine learning can effectively address the shortcomings of inaccurate cardinality estimation of query optimizers. In recent years, a growing number of studies have started to use machine learning or deep model learning to estimate Cardinality [25,26,27,28], and have shown great potential to surpass traditional methods in terms of accuracy. However, limited by some special predicates, unsatisfactory generalization performance on multi-table joins, and special cases of huge deviations between estimated and the

actual values, the research in this area have not been largely used in practical database environments.

Instead, researchers represented by Marcus [29,35] have looked beyond the limitations of cardinality estimation and focused on implementing "learned query optimizers"[29,35,30,31,32,33,34,35], which are machine learning-based models that no longer rely exclusively on traditional cost models. Instead, they use cost as a feature of the model, by learning the relationship between features and runtime, which in turn improves model prediction accuracy. These new learned cost models have been shown to improve the average performance compared to the traditional cost model. However, such learned models require large amounts of training data and hours of training time, and even then, when the schema of the database changes, the models then need to be redesigned and re-trained, so the new learned cost models are too "expensive" compared to traditional cost models. In addition to that, their accuracy can be poor for queries and plans they have not seen in training. As a result, they cannot be reliably used in practice even with large amounts of training data and hours of training time.

In the following, we will analyze and discuss more specifically the work mentioned above.

2.2.1 Application of Statistical Relational Learning in Cardinality Estimation problem

LEO [35] was one of the first attempts to learn from real-world experience to correct errors in cardinality estimation. It used "adjustment factors" to fix errors in cardinality estimation, and these factors was defined as the ratio of the estimated value to the actual cardinality value. The simplistic model fails to take into account other factors such as predicate type and column correlation.

Getoor [36] and Wentao [37] also made some early attempts to use mathematical models to learn the cardinality of base tables and cross-table queries.

Based on a probabilistic graphical model (RPM), a technique that can compactly represent complex joint distributions over a high-dimensional space, Getoor et al. [36] cleverly exploit this joint distribution over multiple attributes, rather than over isolated attributes, to propose a framework for estimating the selectivity of queries in relational databases. In this framework, since Bayesian networks can be used to represent the interactions between attributes in a single table, providing high-quality estimation of the joint distribution over attributes in that table, they utilize the correlations between attributes of tuples connected by foreign keys to extend Bayesian networks to relational domains, thus enabling the estimation of selectivity involving queries over multiple tables. In addition, they have tested and achieved improvements in selective estimation on a number of databases in the medical, financial, and social domains. However, although the probabilistic graphical model can represent complex joint distributions over a high-dimensional space, effectively reduce the constraints on the assumptions of uniformity and independence, such joint distributions are not exactly equivalent to the true distribution of the data. Besides, once the schema of database is changed, the probabilistic graphical model will no longer be accurate and the model will need to be rebuilt. In addition, the database for model testing is too simple and lacks validation of performance on databases containing multiple multi-attribute tables.

Wentao et al. [37] propose a sampling-based method for refine cardinality estimation. Their work is based on PostgreSQL. In PostgreSQL, the cost of an operator (Co) in a query plan can be calculated with the following equation,

$$\mathrm{Co} = N^T \cdot \mathrm{C} = \mathrm{Ns} \cdot \mathrm{Cs} + \mathrm{Nr} \cdot \mathrm{Cr} + \mathrm{Nt} \cdot \mathrm{Ct} + \mathrm{Ni} \cdot \mathrm{Ci} + \mathrm{No} \cdot \mathrm{Co} \quad [2]$$

where PostgreSQL uses 5 parameters to define the cost, i.e. $\mathrm{C} = (\mathrm{Cs}, \mathrm{Cr}, \mathrm{Ct}, \mathrm{Ci}, \mathrm{Co})^{\mathrm{T}}$, and the

meanings of the 5 parameters are shown in the following table,

Parameters	Explanation
Cs	the I/O cost to sequentially access a page
Cr	the I/O cost to randomly access a page
Ct	the CPU cost to process a tuple
Ci	the CPU cost to process a tuple via index access
Co	the CPU cost to perform an operation such as hash or aggregation

Table 1: Parameters of cost and explanation

Similarly, the values N = (Ns, Nr, Nt, Ni, No)^T represent the number of pages sequentially scanned, the number of pages randomly accessed, and so forth.

Through the equation 2, the factors affecting cost can be divided into two major categories, namely C and N. For C, the paper designs five independent queries to calibrate the parameter values of Cs, Cr, Ct, Ci, and Co, respectively. For N, the paper utilizes a sampling-based algorithm to refine cardinality estimation, and the algorithm proceeds as Figure 9.

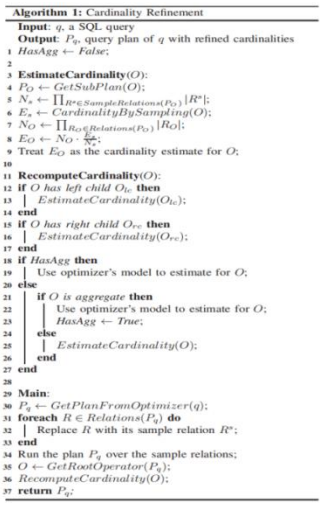


Figure 9: The algorithm of Cardinality refinement [37]

For a given query plan, the cardinality of all operators except Aggregation is refined by two procedures, Recompute Cardinality and Estimate Cardinality. The refined C and N are then used to obtain to more accurate cost of the query plan.

Although this divide-and-conquer algorithm improves the reliability of the traditional cost model to some extent, however, the model does not take into account the potential impact of predicates. In addition, the model cannot refine operator Aggregation's cardinality estimation, and for Aggregation, the model still relies on the database optimizer, which still leads to non-negligible errors in cardinality estimation.

2.2.2 Application of Deep Learning based method in Cardinality Estimation problem

Researchers such as Lakshmi have brought deep learning to the study of cardinality estimation, and their work have shown the great potential of neural networks for cardinality estimation.

Lakshmi et al. [25] pioneered the application of deep learning to cardinality estimation. They proposed a neural network-based approach to learn the selectivity of predicates for user-defined functions (UDF). The structure of their neural network is shown in Figure 10. They transformed complex data objects into feature vectors that could be used for neural network training and then used the features as inputs to a Back-Propagation (BP) neural network to predict selectivity. As an early application of deep learning in cardinality estimation, their attempt is respectable, but their study did not involve selectivity of join predicates and resulting in less attention from the academics.

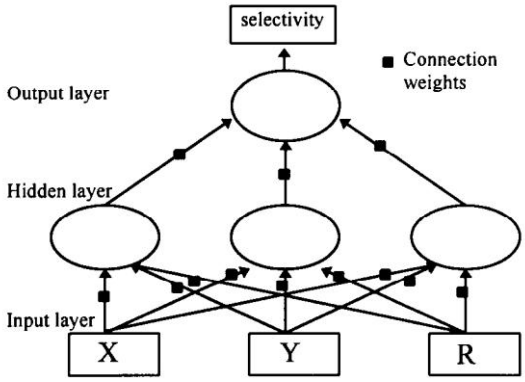


Figure 10: A back-propagation neural network in Lakshimi's paper [25]

Liu et al. [38] treated the cardinality estimation problem as a supervised learning problem by generating synthetic queries for a given table and then using these queries to train a neural network model for predicting the selectivity of queries with range predicates on the base table. In this way, the model is able to perform cold-start training while collecting workload information at runtime that can later be used to refine the model. However, the model does not directly deal with cross-join cardinality estimation and some predicates such as LIKE, IN and IS NOT.

Kipf et al. [26] propose a deep learning based method for predicting cardinality. Join-crossing correlations is a challenge for cardinality estimation, i.e., cardinality estimation is accurate for a single table, while two and more tables are involved, cardinality estimation is routinely wrong by orders of magnitude, which will also lead the optimizer to choose a poor query plan when enumerating the search space and result in a slow query.

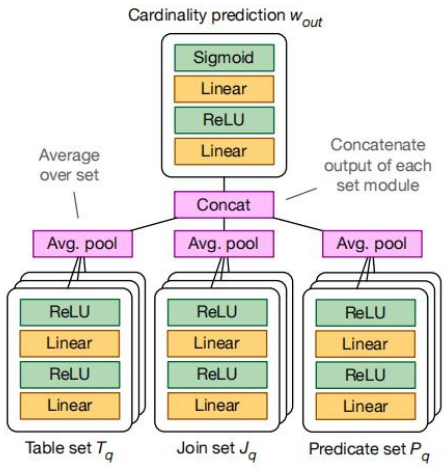


Figure 11: The architecture of the Multi-set Convolutional Network (MSCN) [26]

This paper proposes a novel solution to the challenge. It first extracts features from three different dimensions of the query, i.e., table, joins and predicates, and then feeds these features into a Multi-set Convolutional Network (MSCN) with three independent Multilayer Perceptrons (MLPs) to process these three features respectively. Finally, the output of the three Multilayer Perceptrons (MLPs) are concatenated and fed into an MLP network to predict cardinality (shown in Figure 11).

Although compared to the traditional method, the method based on Multi-set Convolutional Network (MSCN) improve cardinality estimation, it is limited by the number of joins involved in the queries used (up to 4 joins), which hardly prove the model's generalization for queries involving multiple tables. In addition, there are no complex predicates (e.g., LIKE) in the queries, so the model is only applied on the queries with a few joins and simple predicates. Furthermore, the model does not cope well with database changes, i.e., the schema of the database changes the model will likely need to be redesigned and re-trained.

2.2.3 Application of Machine Learning based method in Learned Cost Model

Both the traditional statistical method and the deep learning method for cardinality estimation, are essentially based on the traditional cost model and refine the cost model, while the learned cost model goes beyond the traditional cost model to explore optimal a query plan by a systematic approach of “learning from past failures".

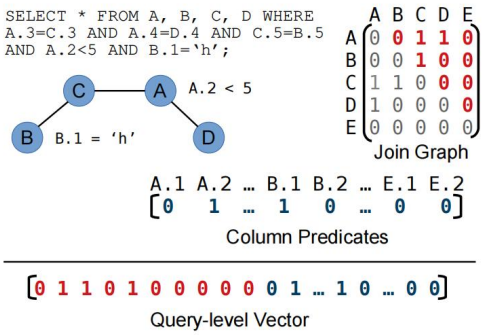


Figure 12: Query-level encoding by Neo [29]

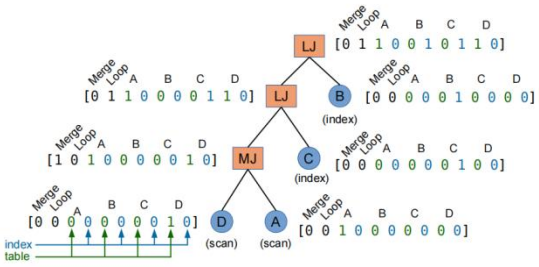


Figure 13: Plan-level encoding by Neo [29]

Neo [29] is a representative of this “new” learned cost model. For a given query, it first extracts features from query level (e.g., the adjacency matrix representing database table-to-table joins)

and plan level (e.g., the type of operator. It utilizes various encoding techniques (e.g., one-hot, hist, etc.) to represent these features respectively (e.g., Figure 12 and Figure 13).

As shown in Figure 14, Neo takes query-level ending as the original input, uses 3 fully connected layers to learn and compress the length of query-level features, and then concatenates the output of fully connected layers with the plan-level encoding (a tree) to generate an augmented tree, as the input of Tree Convolutional Neural Network (TCNN), which is utilized to process tree-shaped data. After the process of TCNN, the resulted tree is flattened into a single vector and fed to another set of fully connected layers to eventually predict the execution time.

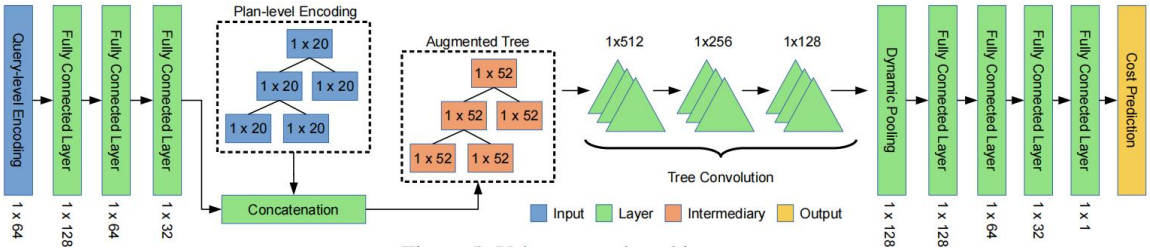


Figure 14: The architecture of Neo [29]

Neo, a novel learned cost model, blurs the boundaries between the main components of traditional query optimizers: cardinality estimation, cost model, and plan search algorithm.

However, Neo is not perfect. For example, the unbalanced way of concatenating the compressed query-level encoding and plan-level encoding will make query-level features play a more significant role in the augmented tree. In addition, Neo is limited by the schema of the data, when a new table is inserted into the database, Neo will face the problem of redesign and re-trained.

Another representation of “new” learned cost model is Bao [23]. Unlike other learned query optimization approaches that must relearn what the traditional query optimizer

already knows, Bao is fully integrated into PostgreSQL as an extension that sits on top of the optimizer and recognizes that the traditional query optimizer contains decades of carefully hand-coded wisdom to enhance query optimization.

Given a query, Bao selects a set of coarse-grained hints that limit the search space of the query optimizer. For the hint, it is the limitation for the operators in the query plan, such as disable hash join and index scan. Via learning to select different hints for different queries, Bao could discover and "steer" query optimizer to generate a relatively good query plan for an incoming query. Specifically, for each incoming query, Bao uses four different hints to generate query plan, which can prune the search space to some extent, but due to the limitation of hints, Bao is likely to exclude the optimal query plan, resulting in it can only generate sub-optimal query plan. However, this does not obscure the merits of Bao. Marcus et al. realized the common problem of learned optimizers, i.e., the inability to adapt to changes in data and workload, and they designed a novel representation for a query plan tree to solve this problem, Figure 15 shows how Bao vectorize a query plan tree. Each node in the query plan is transformed into a vector containing two parts, i.e., a one hot encoding of the operator type and cardinality and cost model information. By using this representation, Bao avoids the need to directly use tables and attributes in the database and does not have to redesign the representation and retrain the model as other learned cost models do when the table or attributes changes, thus improving the generalizability.

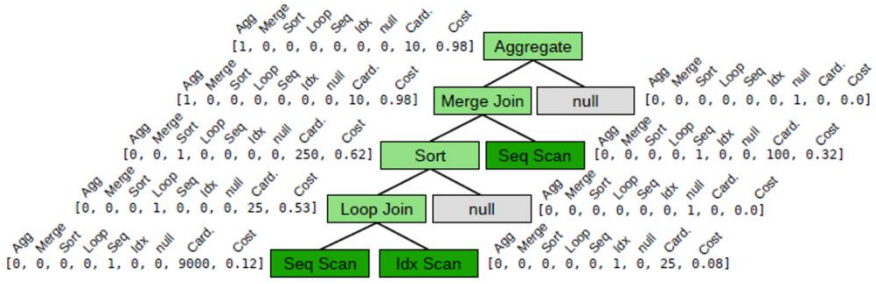


Figure 15: Bao's representation for a query plan tree [23]

Similarly, Guoliang Li et al. [39] proposed an end-to-end tree-structured model for predicting the cost as well as the cardinality of a query plan.

Their model (shown in Figure 16) is divided into three parts. Training Data Generator obtains data by running the workload, the training data is a ternary including <actual execution time of the query plan, actual cost of the query plan, actual cardinality of the query plan>. Feature Extractor extracts and encodes features for each node of the execution plan. With the previous step Feature Extractor, they obtain a tree-structured data where each node is a vector, followed by a Tree-structured Model to learn a neural network model to predict cost and cardinality. Their feature extraction approach considers both query-level and plan-level features, however, the feature extraction and query representation is based on a static specific database, and when the table changes, the query representation may need to be redesigned.

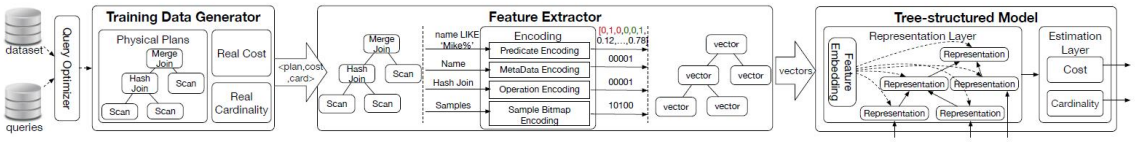


Figure 16: The architecture of learning-based cost estimator [39]

2.2.4 Application of Deep Reinforcement Learning based method in Join Order Enumeration

Finding an optimal join order is one of the most studied problems in the database systems literature [40]. However, exhaustively enumerating and evaluating all possible join orders is too expensive, because N join relations may lead to N plans [40]. Therefore, query optimizers use different strategies to limit their search space, however, pruning the search space can easily result in the negation of the optimal query plan. With the development of machine learning techniques,

methods have emerged that use Deep Reinforcement Learning (DRP) to solve the join order selection problem. First of all, traditional methods are not able to learn from historical experience, while the exploration and exploitation strategy in reinforcement learning is able to both utilize good selection methods from previous execution plans and explore new and potentially better execution plans. Besides, reinforcement learning is almost impossible to exhaust all possible actions and rewards in the join order selection problem, so neural networks are employed to estimate them. The following is the representative work in this filed.

The first representative work is ReJOIN [41], which uses a proximal policy optimization algorithm (Proximal Policy Optimization) to guide the join order selection. The key component is the neural network used for join order selection. The structure of the neural network is shown in Figure 17. The input to this neural network is divided into three parts. The first part is a tree structure vector containing depth information and join information; the second part is the join predicate information appearing in SQL statements; the third part is the table information and column information corresponding to the selective predicates appearing in SQL statements. And the output of the neural network is the probability distribution of each action, through which the next join action is selected. For different SQL statements, there are different neural network parameters as well as rewards, and ReJOIN will estimate the rewards afterward based on these previous reward information, thus enabling the enumeration of the join order of the SQL in the test set.

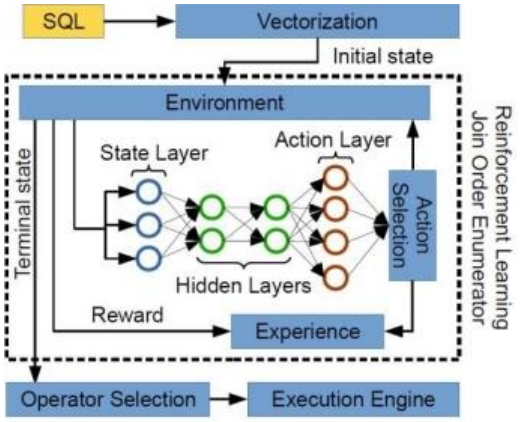


Figure 17: The structure of the neural network in ReJOIN [41]

However, this approach still relies on the optimizer's cost model, which depends on cardinality estimates that are prone to large errors. In addition, the query encoding depends on the database. When the schema of the database changes, it will require the model to be redesigned and retrained, which is expensive.

Another representative work is RTOS [42], which addresses the deficiencies present in ReJOIN. The structure of the RTOS is shown in Figure 18, and the state representation of RTOS is divided into three parts: (1) query information in SQL statements represented by a neural network; (2) table and column information represented by a neural network; and (3) join tree and join state information represented by multiple Tree-LSTM combinations. Through this representation, when a new column needs to be added to the table or a new table needs to be added to the database, directly apply for a new parameter to represent it without retraining, which solves the shortcomings of ReJOIN's inability to cope with changes in the database.

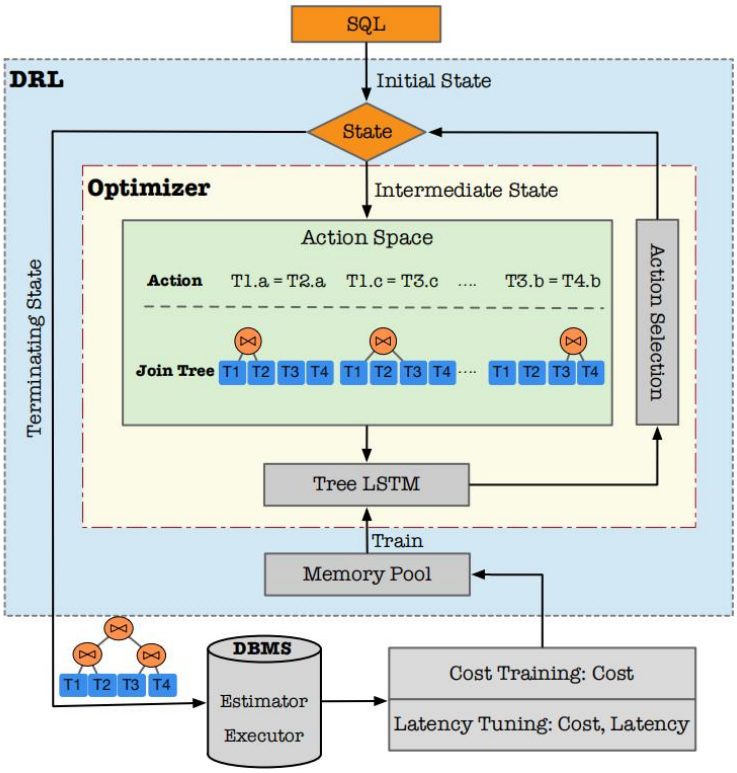


Figure 18: The architecture of RTOS [42]

However, RTOS uses a linear combination of cost and execution time of query to define loss during training. In fact, cost is not an estimate of execution time (cf. Index C), and without figuring out the relationship between cost and execution time, it is questionable to use this combination to define loss.

2.2.5 Comparison of techniques applied to Query Optimization

In the following, we will give a brief overview of the literature mentioned above and compare their advantages and disadvantages.

Category	Producer	Advantage	Disadvantage
Statistical method in cardinality estimation	LEO (Markl et al.)	Novel approach to correct the error in the cardinality estimation	● Fails to take predicate type and column correlation into account
	Getoor et al.	Reduce the constraints on the assumptions of Uniformity and Independence	● The database for model testing is too simple and lacks validation of performance on databases containing multiple multi-attribute tables. ● Once the schema of database is changed, the model will need to be rebuilt.
	Wentao et al.	The refined parameters are helpful to obtain to more accurate cost of the query plan	● Do not take into account the potential impact of predicates. ● Cannot refine some special operator's cardinality estimation, like Aggregation.
Learning based method in cardinality estimation	Lakshmi et al.	Pioneering Introduction of deep learning to the cardinality estimation problem	● Architecture of model is too simple. ● Do not involve selectivity of join predicates.
	Liu et al.	Can perform cold-start training	● Does not directly deal with cross-join cardinality estimation. ● Does not deal with some predicates such as LIKE, IN and IS NOT.
	Kipf et al.	Can provide more accurate cardinality estimation	● Only applied on the queries with a few joins and simple predicates. ● Do not cope well with database changes
Learning based method in learned cost model	Neo (Marcus et al.)	Neo, an end-to-end model, do not need to respectively cope with the main components of traditional query optimizers: cardinality estimation, cost model, and plan search algorithm.	● The unbalanced way of concatenating the query-level features and plan-level features to represent a query is questionable. ● Limited by the schema of the database.
	Bao (Marcus et al.)	● By using hints, effectively prune the search space. ● A novel	● Due to the limitation of hints, Bao is likely to exclude the optimal query plan, resulting in it can only generate sub-optimal query plan.

		representation of query is able to adapt to changes in data and workload	• Do not take into account of query-level features.
	Guoliang Li et al.	Considers both query-level and plan-level features	• Query representation is based on a static specific database. Once the schema of database is changed, the model will need to be rebuilt.
Deep Reinforcement Learning based method in Join Order Enumeration	ReJOIN (Marcus et al.)	Use exploration and exploitation strategy to learn from previous query plan and explore potential optimal query plan	• Relies on the optimizer's cost model which is not always reliable. • Query encoding depends on the database, When the schema of the database changes, the model needs to be redesigned.
	RTOS (Guoliang Li et al.)	Representation can cope with database changes	• The linear combination of cost and execution time to define loss function still needs to be proved.

Table 2: Advantages and disadvantages of the model in the literature

From the Table 2, we can discover that for cardinality estimation, whether using traditional methods (e.g., histogram or sampling-based methods) or learning based methods, they are still essentially dependent on the traditional cost model, and to some degree, these methods are able to reduce the deviation of cardinality estimation from the true value, thus improving the reliability of the traditional cost model. However, due to some constraints, such as real data often not satisfying the underlying assumptions (independence and uniformity), and the lack of validation for queries covering multiple joins and complex predicates, such cardinality estimation-only improvements cannot guarantee that the traditional cost model will not produce a bad query plan.

For learned cost model, they blur the boundaries between the main components of traditional query optimizers: cardinality estimation, cost model, and plan search algorithm. However, due to the representation of query, these models cannot fully and effectively utilize the features of query.

In addition, such learning-based models are often constrained by the schema of database, and how to cope with changes in the data is also a problem we need to consider.

As for join order enumeration problem, comparing to traditional methods, methods based on deep reinforcement learning can learn from historical experience explore new and potentially better query plans with exploration and exploitation strategy. However, the encoding of query and the parameters such as loss function or reward in the training process needs more exploration.

Chapter 3

Introduction of the Hybrid Cost Model

The proposed hybrid model, depicted in Figure 19, comprises two key components: a set of base cost models and a query classifier. The base models are composed of one or more LCMs, along with the CCM. The hybrid cost model aims to forecast the execution time of query plans, leveraging a base cost model that is anticipated to provide better estimations compared to the other base models. The Query Classifier determines this by taking as input the query-level features and predicting the probability that a particular base model is superior to its alternatives. The subsequent chapter elaborate on these components in greater detail.

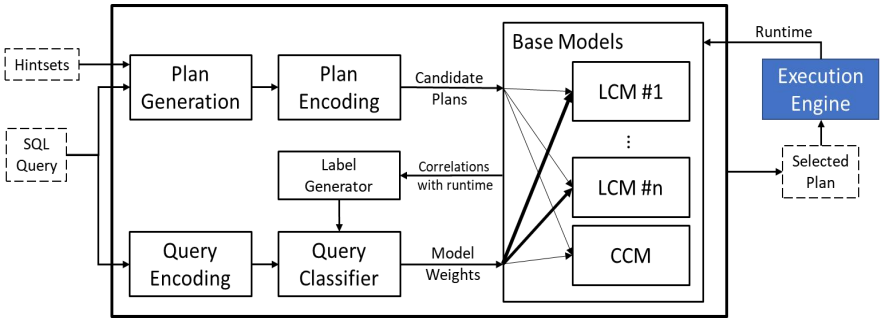


Figure 19: The architecture of the hybrid cost model

3.1 The Learned Cost Models and its training-related modules

The hybrid cost model uses one or more learned cost model(s) along with the classic cost model. Each learned cost model can be trained to predict plan execution time for a certain class of queries. The classes of queries can be defined based on their level of complexity (e.g., number of join and local predicates, aggregations, etc. define the classes) or based on their coverage of the database schema (subsets of the schema define the classes), or application or workload

characteristics or a combination of these factors or other factors. Alternatively, a single model can be trained on all classes of queries. The choice of model granularity comes down to balancing the benefits of model specialization versus the risk of severe over-fitting to a certain class. Figure 20 shows the process to train a learned cost model, and the modules in the training process will be introduced in the following sections in this chapter.

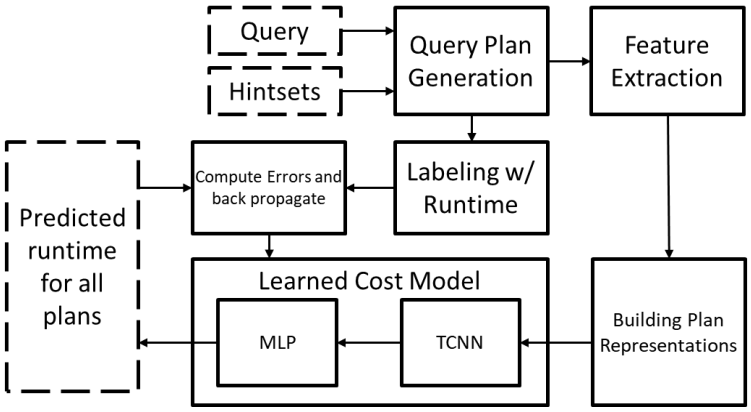


Figure 20: Training procedure for a learned cost model

3.1.1 Plan Generation and Hint Set

For plan generation, it aims at the diversification of query plan. For a given query, we expect to generate all potential query plans so that the model can learn the properties of various query plans, and so that the model can learn to choose a query plan that is closer to the optimal query plan.

There are a variety of techniques for plan diversification, for example, methods such as prompts that enforce certain operators in a plan, random plan generation, or any other technique for plan diversification that can be used for model training.

For this work, it uses hint sets, which are a technique for directionally guiding the query optimizer to generate a query plan by enforcing hints for certain operators in the plan. And Appendix A will offer the detail of the hint sets that model uses.

3.1.2 Plan Encoding

Labeling

After obtaining the query plan for the query, in the next step, each query is executed using each generated plan and the execution time is collected as a label for the learned cost model.

Feature extraction and representation

The next step is to extract the features required to represent the plan tree, for each node of the plan tree we have used the method proposed by Bao [10], and the features we extracted contain two parts. The first part is the type of operator which contains scan operator (e.g., table scan, index scan) and join operator (e.g., merge join, hash join, nested loop join, etc.). The second part is the cardinality and cost of the node.

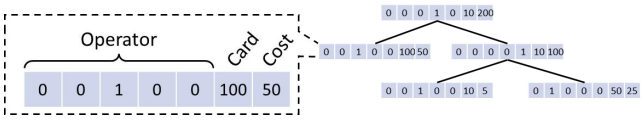


Figure 21: Vectorized query plan tree

By extracting the features of all the nodes and connecting these nodes in the shape of a plan tree, this vectorized tree (shown in Figure 21) is used as a representation of the query plan.

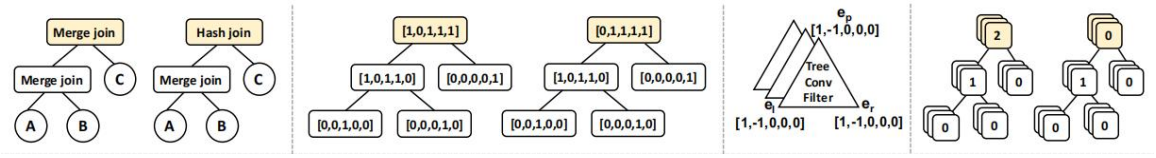
3.1.3 The Learned Cost Models

We train a learned cost model to learn the associations between patterns in the plan trees and runtime using the obtained plan encoding of all the candidate query plans and the execution time of query plans. As shown in the figure 20, the learned cost model consists of TCNN and MLP.

TCNN and MLP

TCNN is a special Convolutional Neural Network. It is utilized to process tree-shaped data.

Figure 22 shows an example of TCNN, Figure 22 - a represents two query plan trees, Figure 22 - b contains operator information for each node of the query plan trees, and Figure 22 - c is a Tree Convolution Filter, the triangle-shaped filter will slide through the tree data. Specifically, the three nodes of the filter are dot-producted with the parent node, left child and right child. Take the node corresponding to table as an example, it performs dot product operation with the node corresponding to filter, i.e., 0 * 1 + 0 * (-1) + 1 * 0 + 0 * 0 + 0 * 0 = 0. This explains why in Figure 21 - d, the data of the node corresponding to table A becomes 0. Thus, for a given node, by sliding the filter in the data, the layer in the network will learn its weights from the node itself, its left child and right child. As such, the parent node will learn the information from its children. After a number of layers, the information from all children is accumulated to the root parent node. The TCNN module takes the vectorized plan tree as input and learns kernels that capture the relationships between the parent nodes and the child nodes. It produces a new vectorized tree. The nodes of this tree are then aggregated into a one-dimensional vector by dynamic pooling. The vector produced by the TCNN module is fed to a Multilayer Perceptron (MLP) which in its final layer predicts the execution time.



(a) (b) (c) (d)

Figure 22: An example of TCNN [29]

3.2 The Query Classifier and its training-related modules

The hybrid cost model also includes a classifier that decides which cost model should be used for planning a given query. This module takes the representation of query-level features as input, which complements the information from plan-level features. Then it predicts which model the query should be planned with. Similarly, its training-related modules will also be introduced in the following sections of this chapter.

3.2.1 Query Encoding

The query encodings are generated using a proprietary method built by the IBM Db2 ML Optimizer team. It uses the structural information of the query's join graph and encodes information about the tables, local predicates, join predicates, aggregations, etc. Therefore, this representation is agnostic to the plan (join orders and plan operators) that will be used to execute the query.

3.2.2 Label Generator

A Label Generator (shown in Figure 8) generates the labels required for the Query Classifier. It collects the predicted execution time from the corresponding LCM(s) and the estimated cost from the CCM. For each query, it computes the Pearson correlation between the estimated value from each base model and the actual runtime. The model with the highest correlation with runtime would be the most suitable for that query. As such, each query is labeled with the base model with maximum Pearson correlation among all models.

3.2.3 Query Classifier

Then, the query representations are produced using the join graph structure. Finally, the query representations and the labels are used to train the Query Classifier that learns to predict the most suitable cost model. The final layer of the Query Classifier uses a SoftMax activation function, which produces the likelihood of superiority of each base model. Therefore, the output values can alternatively be used as weights for combining the predictions from individual models.

Chapter 4

Experimental Setup

4.1 Database and Working Environment

The prototype is built based on IBM Db2 v11.5. The experiments are conducted using the TPC-DS dataset [43]. This dataset aims to represent real-world decision support scenarios and is designed to test the scalability, efficiency, and reliability of decision support systems. The database schema allows for various query patterns such as chain, star, snowflake, and other multi-dimensional query patterns. It contains 7 fact tables, and 17 dimension tables, where each table contains an average of 18 columns. All the experiments involved in the thesis are conducted on IBM servers.

4.2 Query Generation

For most machine learning projects, having a large and processed dataset is essential, but acquiring this data is often a huge challenge. Not only does it mean collecting data from the real world, but it also has to be manually cleaned and labeled. To be able to conduct our experiments effectively, we choose to use synthetic queries, and there are three reasons:

First of all, compared with real data, synthetic data is easy to be generated. We can generate thousands of queries so that machine learning models can better learn the features of the data. However, for real data, it is difficult to have access to a large number of queries to conduct our experiments due to various constraints, such as user privacy. Second, the queries obtained in reality are often simple queries, and it is difficult to cover multiple tables with one query. For

synthetic queries, we can generate more complex queries with multiple tables joins according to the requirements, so that the learned model can better distinguish the features between different queries. Finally, the queries obtained in reality tend to query some popular tables, and it is difficult to cover all the tables in the database. For example, for a movie database, users are mostly interested in the lead actor and director, but not the cost of costumes and props in the movie. Synthetic queries do not have this data "bias" and generate queries that cover a wider range of tables in the database. As mentioned above, for any query, we run all its query plans with the constraint of hint and collect the execution time. To exclude chance, each plan is executed three times, and the average value is used as the final execution time.

For the experiment, we generate 800 valid queries (where the query result is not null) with 1 to 10 equality inner-joins. Besides, for local predicates, in order to running into too many zero-tuple scenarios, we ensure that the number of local predicates in each query is proportional to the number of joins. For example, Figure 23 shows part of the code for generating queries, where nJoins is the number of joins in a query and nLoaclPreds is the number of local predicates in a query. As shown in the code in the figure, for a query with ten joins, it will also have ten local predicates. As for local predicate operators, we use equality and inequality operators including (i.e. ==, <, >, <=, >=).

```
if __name__ == "__main__":

    # Number of joins. Can input a minimum number of joins and a maximum; [minJoins, maxJoins], or a set number of joins i.e: 5
    nJoins = [10,10]

    # Number of queries.
    nQueries = 80

    # Output directory.
    outDir = 'output/'

    # Final output files with filled in skeletons/queries.
    skelFile = 'simpleSkels.txt'
    outputFile = 'simpleQueries.txt'

    # By default (False), always generate one join predicate for each join.
    allowCartesians = False # By default (False), always generate one join predicate for each join.

    # Number of maximum remaining join predicates to be randomly distributed throughout each join.
    # It isn't guaranteed that all of these join preds will be applied because it depends how many
    # join choices there are for each set of tables.
    #
    # Assuming all nMaxAdditionalJoinPreds can be applied:
    #
    # If allowCartesians = True, the total # of join preds is equal to nMaxAdditionalJoinPreds.
    # If allowCartesians = False, the total # of join preds is equal to nMaxAdditionalJoinPreds + nJoins
    nMaxAdditionalJoinPreds = 0

    # Number of local predicates.
    nLocalPreds = 10

    # Add a group by clause to the end of each query using a random column referenced in the query.
```

Figure 23: Part of the code to generate queries

4.3 Plan Generation

We use a technique similar to Bao [23] for generating plans. We adopt the hist sets that are more likely to produce valid plans in Db2. This included 12 hint sets out of a total of 46 hint sets suggested by Bao. The code in Figure 24 shows one of 12 hint sets, with enable Merge Join, Nested Loop Join, Hash Join and Table Scan and disable Index Scan, which directs the optimizer to generate query plan that reads data only through table scan. As such, a given query is compiled using 12 different hint sets to generate 12 query plans. Therefore, the total number of samples used for training and testing an LCM is 9,600 query-plan combinations.

```
query_success_id=0

for i in queries.query_id:
    line = queries[(queries.query_id == i)].query_sql.values[0].strip('\n')
    actual_card = queries[(queries.query_id == i)].actual_card.values[0]
    if "SELECT" in line and i < max_num_queries:
        print("processing query", i+1, "out of", len(queries.query_id))
        command='db2 connect to tpcds;\n'
        command+='db2 .opt set enable mgjn;\n'
        command+='db2 .opt set enable nljn;\n'
        command+='db2 .opt set enable hsjn;\n'
        command+='db2 .opt set disable iscan;\n'
        command+='db2 .opt set enable tscan;\n'
        command+='db2 \"explain plan for ' + line + ';\"\n'
        exp_name = opt_plan_PATH+'query#' + str(query_success_id) + 'CGS.ex'
        command+='db2exfmt -d tpcds -1 -o ' + exp_name + ";\n"
        commands = command.split('\n')
        #for cmd in commands:
            #print('>> ',cmd)
        process = Popen( "/bin/bash", shell=False, universal_newlines=True,
                stdin=PIPE, stdout=PIPE, stderr=PIPE)
        output, _ = process.communicate(command)
        print(output)
```

Figure 24: Code for one of hint sets for generating query plans

4.4 Collection of execution time of query plan

As mentioned above, for any query, we run all its query plans with the constraint of hint and collect the execution time. To exclude chance, each plan is executed three times and the average value is used as the final execution time.

4.5 Data Division

After collecting all the query plans as well as the execution time, we randomly divide the dataset, using 80% of the query for training and 20% of the query for testing. Specifically, we use 7680 query plans (640 queries) for training and 1920 query plans (160 queries) for testing.

4.6 Implementation Details

4.6.1 Preprocessing

The execution time used as labels for training the LCMs exhibits a large skewness. In addition, it can range from milliseconds to several minutes. Therefore, we apply log transformation and min-max scaling respectively. The log transformation reduces the skewness while the min-max scaling brings the scale of the values to a range between zero and one.

4.6.2 The Learned Cost Model

In our prototype, we implemented a single LCM that is trained using the entire training data. The implementation of the LCM consists of two parts, the Tree Convolutional Neural Network

(TCNN) and two fully connected layers. The structure of the LCM is shown in the code in Figure 25. In our experiments, TCNN has 3 layers, and the numbers of channels in each layer are 10, 256, and 128 respectively. Each TCNN layer uses the Rectified Linear Unit (ReLU) for the activation function. The output of the final layer is aggregated to a one-dimensional vector using dynamic pooling. Two fully connected layers are used to take the plan representation produced by the TCNN and predict the execution time of the query plan. The hidden layers consist of 64 cells and 32 cells respectively. Considering that the predicted execution time is a non-negative value, we use Sigmoid for the activation function of the output layer.

```
net = nn.Sequential(
        tcnn.BinaryTreeConv(10, 256),
        tcnn.TreeLayerNorm(),
        tcnn.TreeActivation(nn.ReLU()),
        tcnn.BinaryTreeConv(256, 128),
        tcnn.TreeLayerNorm(),
        tcnn.TreeActivation(nn.ReLU()),
        tcnn.BinaryTreeConv(128, 64),
        tcnn.TreeLayerNorm(),
        tcnn.TreeActivation(nn.ReLU()),
        tcnn.DynamicPooling(),
        nn.Linear(64, 32),
        nn.Sigmoid(),
        nn.Linear(32, 1)
    )
```

Figure 25: Code for the structure of LCM

4.6.3 The Query Classifier

For the query classifier, we use Multi-layer Perceptron (MLP) Classifier (shown in Figure 26), with two hidden layers each with 200 neurons. SoftMax is used as the activation function so that the predicted values for each class correspond to the likelihood of the model's superiority. In a general case with multiple LCMs, multi-class Cross-entropy can be used as the loss function. In this prototype, we used a binary Cross-entropy as the model had to choose between two options only.

```
def mlp_model(n_inputs, n_outputs, depth = 3, width = 200, loss='mse'):
    model = Sequential()
    model.add(Dense(width, input_dim=n_inputs, kernel_initializer='he_uniform', activation='relu'))
    for l in range(depth-1):
        model.add(Dense(width, kernel_initializer='he_uniform', activation='relu'))
    model.add(Dense(n_outputs, activation='softmax'))
    model.compile(loss='binary_crossentropy', optimizer=rmsprop, metrics=['accuracy'])
    return model
```

Figure 26: Code to define MLP Classifier

4.6.4 Training

Finally, for the training phase (shown in figure 27) of the learned cost model and query classifier, we use the Adam optimizer [44] to update the parameters with the default learning rate of 0.001. In addition, we use early stopping [45] to avoid overfitting and Optuna [46] to tune the hyperparameter of the neural networks.

```
np.random.seed(2023)
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(net.parameters(),lr=0.001)
train_losses = []
valid_losses = []
early_stopping = EarlyStopping(patience=20, verbose=True)
for epoch in range(500):
    # Forward pass
    net.train()
    running_loss = 0.0
    pred1_train = net(x_train)
    # Compute Loss
    loss = torch.sqrt(criterion(pred1_train.squeeze(), y_train))
    optimizer.zero_grad()
    print('Epoch {}: train loss: {}'.format(epoch, loss.item()))
    # Backward pass
    loss.backward()
    optimizer.step()
    running_loss += loss.item()
    epoch_loss = running_loss / 6720
    train_losses.append(epoch_loss)

    net.eval()
    running_loss = 0.0
    pred1_validation = net(x_validation)
    loss = torch.sqrt(criterion(pred1_validation.squeeze(), y_validation))
    running_loss += loss.item()
    valid_loss = running_loss / 960
    valid_losses.append(valid_loss)
    early_stopping(valid_loss, net)
    if early_stopping.early_stop:
        print("Early stopping")
        break
print(len(pred1_train))
```

Figure 27: Part of code for training phase

Chapter 5

Experimental Evaluation

In this chapter, we show the improvement of the hybrid model over a single model, either the LCM or the CCM. While we trained a single LCM that is used along with the CCM, this can be generalized to using multiple LCMs as described in the previous chapters which remains as a future work. Besides, we have explored the relationship between cost and execution time, the comparison of query plans generated by LCM and CCM, and labeling methods for Label Generator. we evaluated their results and offered the conclusion.

5.1 Evaluation for the Hybrid Cost Model

5.1.1 Evaluation Process

As shown in Figure 19, given a query in the testing set, the query representation is given as input to the query classifier which in turn outputs the suitable cost model. Then based on the choice of query classifier, the suitable cost model is used to obtain the query plan as the output of the hybrid model.

While getting the output of the hybrid model, the outputs of the base models are needed too for evaluation purposes. As such, the same queries in the testing set are used as the input to each of the base cost models. The execution time of the plans selected by each cost model is recorded as its performance and used to compute the Sub-optimality of the plan selected by each model compared to an alternative.

5.1.2 Hybrid Model vs the Base Models

We use the function provided in Equation 1 to compare the performance of the hybrid model with the CCM and the LCM, respectively. As shown in Figure 28, we find that both the hybrid model and the LCM provide a significant improvement compared to the CCM. While the hybrid model has a marginal improvement compared to LCM, comparing Figure 28-a and Figure 28-b shows that the elimination of regressions has been more significant than the degradation of improvements. Note from these two figures that while the blue bars (improvements) are mostly preserved when switching from the LCM to the Hybrid model, the red bars (regressions) are visibly reduced.

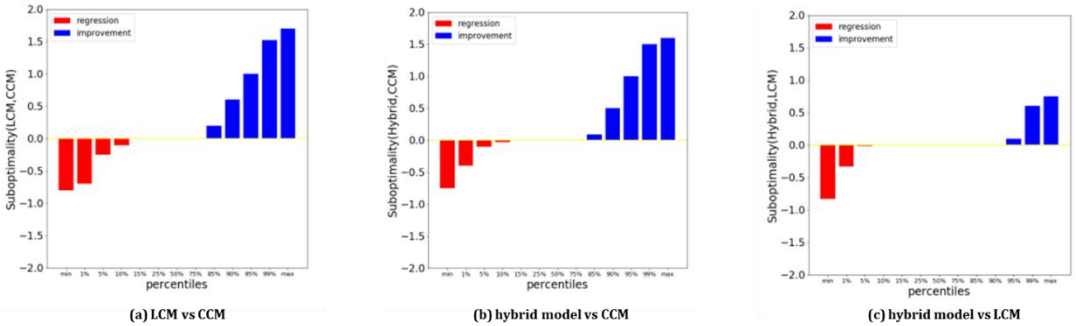


Figure 28: Comparing plan performance results for the Hybrid Model vs. the Base Models

In order to show the difference in the overall performance, drawing upon Equation 1, we introduce the overall suboptimality of an approach compared with a baseline as:

$$\text{SubOpt}(M1, M2) = -\log_{10}\left(\frac{\sum_i (\text{ET}_{M1}(q_i))}{\sum_i (\text{ET}_{M2}(q_i))}\right) \qquad (3)$$

where M1 represents the approach under evaluation, M2 represents the baseline approach, and $\text{ET}_X(q_i)$ represent the execution time of the query q_i using approach X. Values greater than zero represent an overall improvement, while values less than zero represent overall regression. In

addition, we capture the percentage of test samples where an approach improves, degrades, or matches the performance of the baseline. Moreover, we capture the percentage of improvement in the overall performance on the test workload as:

$$\%\ \text{runtime change}\ (M1, M2) = \frac{\sum_i (ET_{M2}(q_i)) - \sum_i (ET_{M1}(q_i))}{\sum_i (ET_{M2}(q_i))} \times 100 \qquad (4)$$

Values greater than zero represent the percentage of improvement compared to the baseline, while values less than zero represent regressions.

Table 3 outlines these values from each of the three comparisons. Note that compared to the LCM, the Hybrid approach reduces the regressions by 10% while it causes only an additional 2.5% new regressions. In addition, the Hybrid approach improves both the total runtime as well as the overall SubOpt values. Therefore, the findings from Figure 11 are supported by the numerical comparison displayed in Table 1, indicating that the performance of the Hybrid model surpasses that of both the LCM and the CCM.

	LCM vs. CCM	Hybrid vs. CCM	Hybrid vs. LCM
SubOpt	0.29591	0.30215	0.00624
% queries improved	33.75 %	31.25%	10%
% queries unchanged	48.75 %	61.25%	87.5%
% queries regressed	17.5%	7.5%	2.5%
% runtime improved	+49.41 %	+50.13 %	+1.43%

Table 3: Comparison Results

5.2 Exploration for the Relationship between Cost and Execution time

5.2.1 Background and Purpose

The traditional cost model is widely used in models based on deep reinforcement learning because of its cheap calculation. Most of these models are based on the optimizer of DBMS and use cost as reward [29,47,48]. Also, cost will be combined with execution time to redefine loss in newly published papers [48]. However, the traditional cost model is not a reliable model. The optimizer will not always generate optimal query plan due to inaccurate cardinality in the cost model, which results in poor query performance. In addition, cost is not an estimate of execution time, so without knowing the relation between cost and execution time, using the combination of them to define loss is debatable. Therefore, we plan to try to explore 2 questions with the help of experiments:

1. Is there a relationship between cost and execution time, or is there a quantitative relationship?
2. Which features may have an impact on cost and execution time, and how important are these features for cost and execution time?

5.2.2 Discussion based on Result

We first compared the cost and execution time between different queries and explored the factors that may affect the cost and execution time. We generated 1000 simple queries (the number of join table is less than 5) and used DB2 optimizer to generate plans. Then the plans were executed, and we record the cost and execution time respectively. Finally, we calculated the Pearson's coefficient of cost and execution time, and the coefficient is 0.037. In contrast, we generated

more complex queries (the number of join table is greater than 5) as input, and the result is 0.020. The results show that there is no strong linear correlation between them.

	Simple queries	Complex queries
Pearson's coefficient between cost and execution time	0.03	0.02

Table 4: The co-relation between cost and execution time

Although cost and execution time do not have a strong linear relationship, we still tried to explore which features may affect cost and execution time. Since it was the initial stage of the experiment, we collected some obvious features of queries, such as cardinality, number of joining tables, number of predicates (join predicates and local predicates) and attempted to study the influence of these factors. The results show that there is a linear relationship between the number of joining tables and the number of predicates (cycle join was not be considered), so we only needed to explore the relation among cardinality, the number of joining tables, cost and execution time, and the results was shown in the table below. (The value in the Table 5 is Pearson coefficient)

	Cardinality	The number of tables (the number of predicates)
Cost	0.45	0.35
Execution time	0.06	0.25

Table 5: The impact of query-level feature for cost and execution time

The results show that for queries involving more tables, cardinality has a greater impact on cost and a relatively smaller influence on execution time, while the number of join table or the number of predicates will have a greater impact on execution time comparing to cardinality.

Since cardinality has a great impact on cost, if we adjust the data for cardinality, the correlation between cost and execution time may change. And when I try to remove some queries with large cardinality and then recalculate the correlation between cost and execution time. we find the correlation coefficient has increased significantly. The results are shown in the Table 6.

	Queries (cardinality < 20000)	Queries (cardinality < 100)
Pearson's coefficient between cost and execution time	0.05	0.068

Table 6: The impact of cardinality for Pearson's coefficient between cost and execution time

The previous experiment mainly focused on different queries, trying to explore the impact of the features of query itself, but the optimizer will parse an incoming query and convert it into a query plan, so the impact of query plan is also important. As for a query plan, it can be represented as a plan tree, which consists of a set of operators. So in the next experiment, we tried to make a horizontal comparison, that is, to study the possible query plans for a single query and then steer the optimizer to generate promising plans by constraining the operator set. After that, we recorded the cost and execution time for each query so that we could explore the impact of operators on cost and execution time. For the experiment, we reused the hint sets of BAO (the operators for join {merge join, harsh join, nested looped join}, the operators for scan {index, sequential}) [23]. And for each query, we generated six plans (one plan was generated by DB2 optimizer without any constraints, and the remaining plans were steered by BAO's hint sets). The experiment will be carried out for simple and complex queries respectively. The experimental results show that for simple queries, though we hoped to focus on join and did not clear the cache after executing each query, the type of scan operators still plays the most significant role in the impact of cost and execution time. In contrast, the effect of the type of join operators appears to be negligible, but for more complex queries, the type of join operators, especially hash join, begins to affect the cost

and execution time, and the execution time of some plans with BAO's hint set [23] is less than that of plans generated by the optimizer.

5.2.3 Conclusion

The experimental results show that there is no strong linear correlation between cost and execution time before, so cost is not a prediction or a substitute for execution time. Among the query-level features that affect cost and execution time, cardinality and number of join or predicate in the query both have a critical impact on cost, while for execution time, number of join or predicate plays a more important role in the execution time. For plan-level features, when the query involves fewer tables (the number of joins is less than 5), the impact of scan type of operators on cost and execution time is still the most important, and for more complex queries (the number of joins is greater than 5), the impact of join type of operators become more important.

5.3 Exploration for the Comparison of Query Plans generated by LCM and CCM

5.3.1 Experiment setup and Result

We generated a set of 320 queries for exploring the CCM and LCM, we used the hint set in Appendix B to diversify our query plans, for each query, we could generate 12 plans, and then used a single LCM with the CCM to predict the predicted execution time of each plan. Then we run all query plans and collect the execution time, predicted execution time and cost of each query plan, and mark three special query plans of each query according to the definition in Table

7.

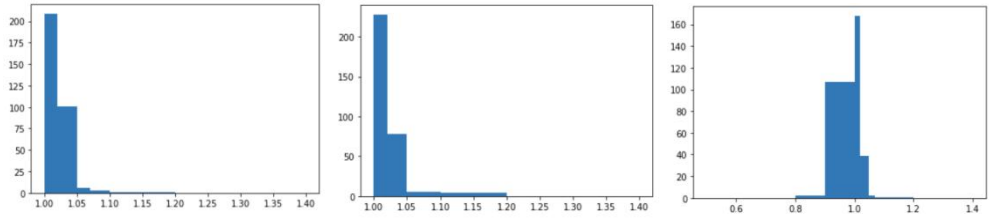
Special plans	Definition
CCM_plan	the plan of a query with minimum cost.
LCM_plan	the plan of a query with minimum predicted execution time.
Best_plan	the plan of a query with minimum true execution time.

Table 7: Definition of three special plans

We compare the characteristics of the query plan of CCM and LCM with Equation 5.

$$\text{Comparison}(P1, P2) = EX(P1)/EX(P2) \quad (5)$$

For example, given query 1, the execution time of CCM_plan is 0.533, and the execution time of Best_plan is 0.524, so Comparison (CCM_plan, Best_plan) equals to 1.015(0.533/0.524). Then we collect Comparisons of all the queries and obtained its distribution (shown in Figure 29), with the x-axis being the Comparisons and the y-axis being the number of queries in the corresponding Comparison interval.



(a)CCM_plan vs Best_plan (b)LCM_plan vs Best_plan (c)LCM_plan vs CCM_plan

Figure 29: The distribution of Comparison

5.3.3 Conclusion

From the similar distributions in Figure 29 - a and Figure29 - b, we can find that most of the data are distributed in [1, 1.05], indicating that in most cases (close to 95%), both CCM and LCM can produce query plans close to the Optimal query plan. And comparing with Compare (CCM_plan, Best_plan), more data of Compare (LCM_plan, Best_plan) are closer to 1, which means LCM can choose a better query plan in many cases.

From Figure 28 - c, we find for 47.5% queries, LCM can choose a better query plan than CCM, and for 34.0625% queries, the query plans of CCM are better. This further corroborates the conclusion that not any singe cost model is suitable for all the scenarios, thus a hybrid cost model can be a better solution.

5.4 Exploration for the Labeling methods for Label Generator

5.4.1 Comparison of three labeling methods

For the Label Generator in Figure 18, whose task is to generate the labels needed for the query classifier, three different methods are explored to achieve this task. (Because of the early exploratory stage of the experiment, we reuse the data from Exploration for the Comparison of Query Plans generated by LCM and CCM in order to improve the efficiency of the experiment.)

Methods 1: Labeling with "better"

```
If Execution time(LCM_plan) < Execution time(CCM_plan):
        Labeling with LCM
Else:
        Labeling with CCM
```

Figure 30: Pseudocode for labeling with "better"

As shown in Figure 30 this approach is relatively straightforward, i.e., directly comparing the execution time of LCM_plan and CCM_plan of query, and then labeling with "better" (less execution time) cost model.

Methods 2: Labeling with Q-error

```
If Q-error[Execution time(LCM_plan), Execution time(Best_plan)]
< Q-error[Execution time(CCM_plan), Execution time(Best_plan)]
        Labeling with LCM
Else:
        Labeling with CCM
```

Figure 31: Pseudocode for labeling with Q-error

As the Figure 31 shows that this method compares the Q-error of the execution time of query plan of cost model and best query plan, and then uses the cost model with smaller Q-error to label a query.

Methods 3: Labeling with Pearson's coefficient

```
If Pearson's coefficient(predicted execution time of query plans, execution time of query
plans)   > correlation(cost of query plans, execution time of query plans):
        Labeling with LCM
Else:
        Labeling with CCM
```

Figure 32: Pseudocode for labeling with Pearson's coefficient

As shown in Figure 32, this approach is done by respectively comparing Pearson's correlation of the actual execution time with the predicted execution time and cost of query plan. The cost model with the higher correlation with the actual execution time will be more suitable for that query. Therefore, each query is marked as the cost model that has the maximum Pearson's coefficient with the actual execution time.

5.4.2 Result and Evaluation

Then we use each of the three methods to generate the required labels for the query classifier and train the query classifier, and then we use accuracy, precision, recall and F1-score to evaluate the query classifier, the following is the result:

Labeling methods	accuracy	precision	recall	F1-score
Labeling with "better"	0.600	0.639	0.650	0.598
Labeling with Q-error	0.600	0.639	0.650	0.598
Labeling with Pearson's coefficient	0.667	0.643	0.667	0.641

Table 8: Comparison of different labeling methods

As shown in Table 8, method 1 and method 2 have the same experimental results, and the result of method 3 is better than the other two methods, so we generate labels with the method based on Pearson's coefficient.

Chapter 6

Conclusion and Future work

6.1 Conclusion

Query optimization has been an important research direction in the field of databases, and many traditional and emerging deep learning methods have been applied to this field with remarkable results. We examined in a comprehensive manner the existing research literature that uses machine and deep learning to tackle various parts of the query optimization process and we discussed their strengths and weaknesses. Ultimately, we focused on the classical cost model (CCM), on which the Optimizer of every DBMS relies in order to estimate the cost of alternative query plans, and devised a novel version of it, the Learned Cost Model (LCM), which learns the query plan costs using machine learning models. Our study showed that although the LCM had a better average performance than CCM, the LCM cannot completely replace the CCM, especially when testing queries do not closely resemble the examples present in the model's training data. Consequently, we proposed a technique that consistently selects a suitable query plan, utilizing both the CCM and the LCMs, which we call the hybrid cost model. The later capitalizes on the advantages of both LCMs and the CCM. Upon receiving a query, the hybrid cost model selects the most appropriate cost model(s) for planning. It learns to rely on a base model when it is expected to deliver better performance than the alternatives.

The hybrid cost model mainly consists of the LCMs, the CCM and the Query Classifier. The LCMs is a set of base learned cost models. The hybrid cost model aims to forecast the execution time of query plans, leveraging a base cost model that is anticipated to provide better estimations compared to the other base models. The Query Classifier determines this by taking as input the

query-level features and predicting the probability that a particular base model is superior to its alternatives.

In the process of developing the hybrid cost model, we encountered problems such as labeling and diversifying the training data. In order to overcome these challenges, we did a series of exploratory experiments, such as exploring the effects of different labeling methods on the model performance and combining the evaluation results to determine the correlation between the estimates and runtime as the better labeling methods. For data diversity, we referred to the solutions in the literature [23], such as the use of hint set to guide the optimizer directed to generate query plan. We expand the hint set on the basis of the solutions in the literature, and the expanded hint set takes into account the possibility of the combination of all operators under DB2 and solves the shortcomings of the literature [23] that the optimal query plan may be excluded because of the use of hint set.

Finally, we conducted a series of experiments to optimize the hybrid cost model and compare the results of the hybrid cost model with those of the single model in different dimensions. The experimental results show that the hybrid approach improves both the total runtime as well as the overall SubOpt values, and the conclusions from the experiments are validated by different dimensions, such as numerical comparison and the distribution of SubOpt, indicating that the performance of the hybrid cost model surpasses that of both the LCM and the CCM.

6.2 Future work

In the future, we plan to continue the research described in this thesis by following four directions. First, we will explore thoroughly the possibility to use multiple LCMs working together. Our goal is to develop different LCMs, so that each one is specialized, i.e., trained, on a specific category of queries and/or query plans. Our utmost goal is to combine the results of multiple LCMs, e.g.,

based on the correlation between the incoming query and each LCM, multiply the correlation with the result of the corresponding LCM, and then sum up all the results to be the final output.

Second, we will explore the model's generalization to out-of-distribution data. In this thesis we have already verified that the model has promising performance on in-distribution data. However, in order to apply the hybrid model into a production environment, the model still needs to be generalized to data that is not following the distribution of the data on which it was trained. One possible improvement is to use unsupervised learning, such as clustering, but since our current representation of query are too long for clustering to be effective, we can consider using a combination of Auto-Encoder and Graph Neural Network to obtain compressed features.

Third, we will explore the model for different databases and different DBMS. The hybrid model is currently developed based on the IBM DB2 DBMS environment using TPC-DS as the experimental database. In order to validate the model's performance in different DBMS environments and databases, we intend to port the model to PostgreSQL and use more databases, such as IMDB [52], to evaluate the model.

Finally, we will explore the self-learning ability of the model. Since the hybrid model is trained in off-line state, we would like to try to learn new queries in on-line state in the next experiments. This involves optimizing some of the training mechanisms, such as when the model needs to be trained again and whether the re-training is based on the original model. We will explore these mechanisms in the following experiments.

Bibliography

[1] Amin K, Verena K, et al. Robust Query Optimization using Probabilistic Machine Learning. Submitted for publication, 2024.

[2] Encyclopedia of database systems[M]. New York, NY, USA:: Springer, 2009: 506-511.

[3] Leis V, Gubichev A, Mirchev A, et al. How good are query optimizers, really?[J]. Proceedings of the VLDB Endowment, 2015, 9(3): 204-215.

[4] Leis V, Radke B, Gubichev A, et al. Cardinality Estimation Done Right: Index-Based Join Sampling[C] Cidr. 2017: 8-11.

[5] Wang X, Qu C, Wu W, et al. Are we ready for learned cardinality estimation?[J]. Proceedings of the VLDB Endowment, 2021, 14(9): 1640-1654.

[6] K-ModesClustering.https://medium.com/@shailja.nitp2013/k-modesclustering-ef6d9ef06449 [Online; accessed 14.03.2023].

[7] sparkbyexamples.com/machine-learning/clustering-in-machine-learning/ [Online; accessed 14.03.2023].

[8] Zacarias-Morales N, Pancardo P, Hernández-Nolasco J A, et al. Attention-inspired artificial neural networks for speech processing: A systematic review[J]. Symmetry, 2021, 13(2): 214.

[9] Bojnordi E, Mousavirad S J, Pedram M, et al. Improving the Generalisation Ability of Neural Networks Using a Lévy Flight Distribution Algorithm for Classification Problems[J]. New Generation Computing, 2023, 41(2): 225-242.

[10] Liu Y, Cao B, Li H. Improving ant colony optimization algorithm with epsilon greedy and Levy flight[J]. Complex & Intelligent Systems, 2021, 7: 1711-1722.

[11] Sammut C, Webb G I. Encyclopedia of machine learning and data mining[M]. Springer Publishing Company, Incorporated, 2017:8.

[12] Learn the essentials of Reinforcement Learning. https://towardsdatascience.com/reinforcement-learning-101-e24b50e1d292 [Online; accessed 22.06.2023].

[13] The Basics of Recurrent Neural Networks. https://pub.towardsai.net/whirlwind-tour-of-rnns-a11effb7808f [Online; accessed 14.03.2023].

[14] Pavlatos C, Makris E, Fotis G, et al. Utilization of Artificial Neural Networks for Precise Electrical Load Prediction[J]. Technologies, 2023, 11(3): 70.

[15] Ganchev T D, Parsopoulos K E, Vrahatis M N, et al. Partially connected locally recurrent probabilistic neural networks[M]. INTECH Open Access Publisher, 2008: 377-400.

[16] What is Long Short Term Memory (LSTM). https://www.knowledgehut.com/blog/web-development/long-short-term-memory [Online; accessed 15.04.2023].

[17] Hochreiter S, Schmidhuber J. Long short-term memory[J]. Neural computation, 1997, 9(8): 1735-1780.

[18] Cho K, Van Merriënboer B, Gulcehre C, et al. Learning phrase representations using RNN encoder-decoder for statistical machine translation[J]. Computer Science, 2014: 1724-1734.

[19] Papineni K, Roukos S, Ward T, et al. Bleu: a method for automatic evaluation of machine translation[C]. Proceedings of the 40th annual meeting of the Association for Computational Linguistics. 2002: 311-318.

[20] Transformers: An Overview of the Most Novel AI Architecture. https://towardsdatascience.com/transformers-an-overview-of-the-most-novel-ai-architecture-cdd7961eef84 [Online; accessed 24.06.2023].

[21] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need[J]. Advances in neural information processing systems, 2017, 30: 6000 - 6010.

[22] Shao Y, Cheng Y, Nelson S J, et al. Hybrid Value-Aware Transformer Architecture for Joint Learning from Longitudinal and Non-Longitudinal Clinical Data[J]. Journal of personalized medicine, 2023, 13(7): 1070-1070.

[23] Marcus R, Negi P, Mao H, et al. Bao: Making learned query optimization practical[C]. Proceedings of the 2021 International Conference on Management of Data. 2021: 1275-1288.

[24] Amin K. Experimental methodologies for applying artificial intelligence techniques in the field of query optimization. Report for Comprehensive Exam, UOttawa, 2022.

[25] Lakshminarayanan B, Pritzel A, Blundell C. Simple and scalable predictive uncertainty estimation using deep ensembles[J]. Advances in neural information processing systems, 2017, 30: 6405 - 6416.

[26] Kipf A, Kipf T, Radke B, et al. Learned cardinalities: Estimating correlated joins with deep learning[J]. Biennal Conference on Innovative Data Systems Research (CIDR 2019). 2019: 9-16.

[27] Liu H, Xu M, Yu Z, et al. Cardinality estimation using neural networks[C]. Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering. 2015: 53-59.

[28] Wu C, Jindal A, Amizadeh S, et al. Towards a learning optimizer for shared clouds[J]. Proceedings of the VLDB Endowment, 2018, 12(3): 210-222.

[29] Marcus R, Negi P, Mao H, et al. Neo: A learned query optimizer[J]. Proceedings of the Vldb Endowment, 2019, 12(11): 1705–1718.

[30] Yu X, Chai C, Li G, et al. Cost-based or Learning-based? A Hybrid Query Optimizer for Query Plan Selection[J]. Proceedings of the VLDB Endowment, 2022, 15(13): 3924-3936.

[31] Sun L, Ji T, Li C, et al. DeepO: A Learned Query Optimizer[C]//Proceedings of the 2022 International Conference on Management of Data. 2022: 2421-2424.

[32] Markl V. Making Learned Query Optimization Practical: A Technical Perspective[J]. ACM SIGMOD Record, 2022, 51(1): 5-5.

[33] Akdere M, Çetintemel U, Riondato M, et al. Learning-based query performance modeling and prediction[C]. 2012 IEEE 28th International Conference on Data Engineering. IEEE, 2012: 390-401.

[34] Marcus R. Learned Query Superoptimization[J]. Joint Workshops at 49th International Conference on Very Large Data Bases (VLDBW'23), 2023, 3462.

[35] Markl V, Lohman G M, Raman V. LEO: An autonomic query optimizer for DB2[J]. IBM Systems Journal, 2003, 42(1): 98-106.

[36] Getoor L, Taskar B, Koller D. Selectivity estimation using probabilistic models[C]. Proceedings of the 2001 ACM SIGMOD international conference on Management of data. 2001: 461-472.

[37] Wu W, Chi Y, Zhu S, et al. Predicting query execution time: Are optimizer cost models really unusable?[C]. 2013 IEEE 29th International Conference on Data Engineering (ICDE). IEEE, 2013: 1081-1092.

[38] Liu H, Xu M, Yu Z, et al. Cardinality estimation using neural networks[C]. Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering. 2015: 53-59.

[39] Sun J, Li G. An end-to-end learning-based cost estimator[J]. Proceedings of the VLDB Endowment, 2019, 13(3):307-319.

[40] Ono K, Lohman G M. Measuring the Complexity of Join Enumeration in Query Optimization[C]. VLDB. 1990, 97: 314-325.

[41] Marcus R, Papaemmanouil O. Deep reinforcement learning for join order enumeration[C]. Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management. 2018: 1-4.

[42] Yu X, Li G, Chai C, et al. Reinforcement learning with tree-lstm for join order selection[C]//2020 IEEE 36th International Conference on Data Engineering (ICDE). IEEE, 2020: 1297-1308.

[43] Nambiar R O, Poess M. The Making of TPC-DS[C]. VLDB. 2006, 6: 1049-1058.

[44] Kingma D P, Ba J. Adam: A method for stochastic optimization[J]. International Conference on Learning Representations, 2015.

[45] Caruana R, Lawrence S, Giles C. Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping[J]. Advances in neural information processing systems, 2000, 13: 402-408.

[46] Akiba T, Sano S, Yanase T, et al. Optuna: A next-generation hyperparameter optimization framework[C]. Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining. 2019: 2623-2631.

[47] Krishnan S, Yang Z, Goldberg K, et al. Learning to optimize join queries with deep reinforcement learning[J]. arXiv preprint arXiv:1808.03196, 2018.

[48] Yu X, Li G, Chai C, et al. Reinforcement learning with tree-lstm for join order selection[C]. 2020 IEEE 36th International Conference on Data Engineering (ICDE). IEEE, 2020: 1297-1308.

[49] Ning Wang, Seyed Mohammad Amin Kamali, Verena Kantere, Calisto Zuzarte, Vincent Corvinelli, Brandon Frendo, Stephen Donoghue. A Hybrid Cost Model for Evaluating Query Execution Plans. In IEEE AIKE, Paper, 2023-09-22.

[50] Seyed Mohammad Amin Kamali, Ning Wang, Verena Kantere, Calisto Zuzarte, Brandon Frendo, Stephen Donoghue.Workload-driven query planning and optimization using machine learning. In In IBM CASCONxEVOKE Exhibition, Poster, 2021-11-23.

[51] Seyed Mohammad Amin Kamali, Vincent Corvinelli, Calisto Zuzarte, Brandon Frendo, Verena Kantere, Ning Wang. Hybrid cost model for query execution plan evaluation, Submitted Patent, 2023-11-15.

[52] Burch M, Baulig G, Boley T, et al. IMDb Explorer: visual exploration of a movie database[C]//Proceedings of the 11th International Symposium on Visual Information Communication and Interaction. 2018: 88-91.

Appendix A

Assumptions of Independence and Uniformity

Traditional methods for Cardinality estimation generally follows two assumptions [26].

Independence: The first common assumption is the attribute-value independence assumption, in the assumption, distributions of individual attributes are independent of each other, and the joint distribution is the product of the single-attribute distributions.

Uniformity: The second common assumption is the join uniformity assumption, which assumes that one tuple from one relation is equally likely to be joined to any tuple from a second relation.

Appendix B

Hint Sets used for Plan Generation

Upon using each of the following hint sets, the operators included in the set are enabled and other operators are disabled:

Hint 1: {merge join, nested loop join, hash join, index scan, table scan}

Hint 2: {merge join, nested loop join, hash join, table scan}

Hint 3: {merge join, nested loop join, hash join, index scan}

Hint 4: {nested loop join, hash join, table scan}

Hint 5: {nested loop join, hash join, index scan}

Hint 6: {nested loop join, hash join, index scan, table scan}

Hint 7: {merge join, nested loop join, table scan}

Hint 8: {merge join, nested loop join, index scan}

Hint 9: {merge join, nested loop join, index scan, table scan}

Hint 10: {nested loop join, table scan}

Hint 11: {nested loop join, index scan}

Hint 12: {nested loop join, index scan, table scan}