Query Optimization with Deep Learning Architectures

by

Theodoros Goulas

Submitted
in partial fulfillment of the requirements for the degree of

Master of Artificial Intelligence

at the

UNIVERSITY OF PIRAEUS

June 2022

Author: Theodoros Goulas

II-MSc "Artificial Intelligence"

June 20, 2022

Certified by. . . . .

Stasinos Konstantopoulos
Post-Doctoral Researcher
Thesis Supervisor

Certified by. . . . .

Antonis Troumpoukis Post-
Doctoral Researcher
Member of Examination
Committee

Certified by. . . . .

Thanasis Vergoulis
Post-Doctoral Researcher
Member of Examination
Committee

Query Optimization with Deep Learning Architectures

By

Theodoros Goulas

Submitted to the II-MSc "Artificial Intelligence" on June 20, 2022, in partial fulfillment of the
requirements for the MSc degree

Abstract

The increasing trend of moving from the old-fashioned centralized database systems into distributed ones significantly increased the query optimization problem's complexity, leading to complicated optimization algorithms based on time and resource-consuming analytical methods. This study proposes introducing natural language processing techniques combined with Deep Learning architectures as a statistical alternative to the traditional analytical query optimization approach to address this issue.

Thesis Supervisor: Stasinos Konstantopoulos
Title: Query Optimization with Deep Learning Architectures

Acknowledgments

First and foremost, I would like to thank the thesis supervisor Mr. Stasinos Konstantopoulos who guided me throughout this project. Besides, I would like to thank the members of the Examination Committee, Mr. Antonis Troumpoukis and Mr. Thanasis Vergoulis, for their significant contribution to the completion of the current study.

Also, I would like to thank my girlfriend Maria and my family and friends for their support. Without that support, I couldn't have succeeded in completing this project.

Contents

List of Figures

List of Tables

1 Introduction

The amount of data published on the Web and the number of data sources have been exploding recently, covering diverse domains. Therefore, applying optimization techniques to the systems querying these data is heavily required. Declarative query languages allow easy expression of complex queries without knowing about the details of the physical data organization of the database. Advanced query processing technology transforms high-level queries into efficient lower-level query execution strategies. The query transformation should achieve both correctness and efficiency. The main difficulty is achieving efficiency, which is also one of the essential tasks of any database management system.

Furthermore, there is a growing trend of moving towards a service-oriented architecture by putting the traditional databases behind web services. As a result, data is not stored in databases bound to a single system but instead is being made available via web services. However, the new structure may increase the versatility but brings additional complexity to the system, affecting the efficiency of the old-fashioned query optimizers.

Efficient query processing over distributed web services in a transparent and integrated fashion demands appropriate manipulation of the individual endpoints. The scientific domain involved with these operations' research and development is Federated Query Processing (FQP). FQP's primary goal is transforming the initial (federated) query over the whole schema into an equivalent set of sub-queries over the distributed databases (local query). As a result, federated query processing is divided into four distinct sub-processes: (i) creating distributed databases representation, (ii) federated query decomposition and proper schemata selection, (iii) execution plan development and optimization, and (iv) query execution, as shown at the following figure.

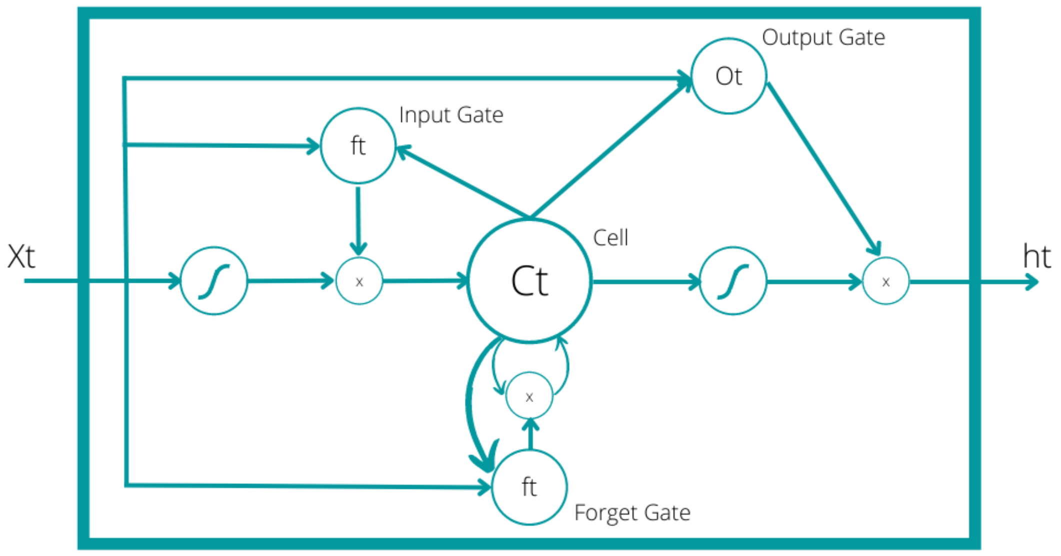

Figure 1: FQP sub-processes

Different approaches have been studied to optimize query processing, leading to numerous corresponding processors. FedX processor utilizes schema representation techniques during the query execution phase (on-flight) and chooses the proper endpoints, accelerating the execution process. On the other hand, the Semagrow processor uses complex and time-consuming optimization algorithms and calculates the optimal execution plan based on the knowledge of data distribution over remote endpoints. As a result, the time consumed during execution plan designing and optimization is counterbalanced by the reduced query execution time. The common attribute of all the algorithms mentioned above is the increased complexity and the enormous computational resource demands. Therefore, a model will be proposed in this project's scope, approximating the optimal execution plans and thus accelerating the whole optimization process using machine learning techniques.

Based on the concept that both the input SQL query and the output execution plan are text sequences, the whole optimization process could be faced as a well-known sequence-to-sequence learning problem. As a result, Neural Machine Translation (NTM) deep neural learning architectures will be incorporated as the prediction generation unit. At the same time, natural language processing techniques will be used to transform the text sentences into suitable inputs for the NTM model, i.e., numeric sequences.

The utilization of the pre-trained NTM model in the optimization process is expected to speed up the query optimization process due to the deep learning models' reduced prediction time while maintaining the whole process efficiency compared to the traditional analytical optimizers.

1.1 Outline

The remainder of this thesis is organized as follows:

- In Chapter 2, background on query optimization and machine learning algorithms.
- In Chapter 3, methodology and implementation technical details are described.
- In Chapter 4, the conducted experiments alongside the individual results are presented.
- In Chapter 5, the whole project's conclusions are discussed.

2 Background

Query optimization is a highly complicated process consisting of several sub-modules, each of whom is responsible for a specific task: (i) the query parser, (ii) the query optimizer, (iii) the code generator, and (iv) the query processor. The complexity of each module raises significantly when distributed database systems replace the traditional centralized ones, diminishing the efficiency of the typical optimizers.

Based on this assumption, it will be attempted to replace the analytical algorithms with statistical ones by introducing deep neural network architectures and natural language processing techniques in the optimization process.

The abovementioned concepts will be thoroughly discussed and analyzed in the following sections.

2.1 Query Optimization

A vital component of every Database management system (DBMS) is the query optimizer regarding the query evaluation process. The query optimizer is responsible for determining the most efficient execution plan for any given SQL query by estimating the costs of every project within the space of possible execution plans. Since the algebraic representation of a SQL query can be transformed into a set of equivalent expressions, the task of the query optimizer is nontrivial [1].

Having stated the above, each query should follow a specific traversal through a DBMS to be answered, as shown in Figure 2.

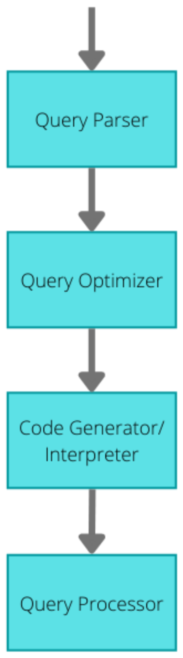

Figure 2: Query flow

The individual modules, as depicted above, have the following functionalities:

- The **Query Parser** validates the query and then transforms it into an equivalent internal form, using relational calculus expressions.
- The **Query Optimizer** generates an efficient execution plan by examining all the equivalent algebraic expressions produced by the query parser and choosing the optimal one.
- The **Code Generator** or the **Interpreter** converts the optimal execution plan into a set of the appropriate calls to the query processor.
- The **Query Processor** executes the query and produces the final output.

Since the core of this project mainly concerns the query optimization process, the following sections will be focused on the modular architecture and functionality of a typical query optimizer [2].

2.1.1 Query Optimizer Architecture

Given a query on a DBMS, a set of equivalent execution plans could be produced. The query optimizer should evaluate all the alternatives to conclude with the most efficient one. The process of generating and examining these alternatives could be generalized to an abstract model, indicating the modular architecture of a typical query optimizer, as shown in Figure 3.

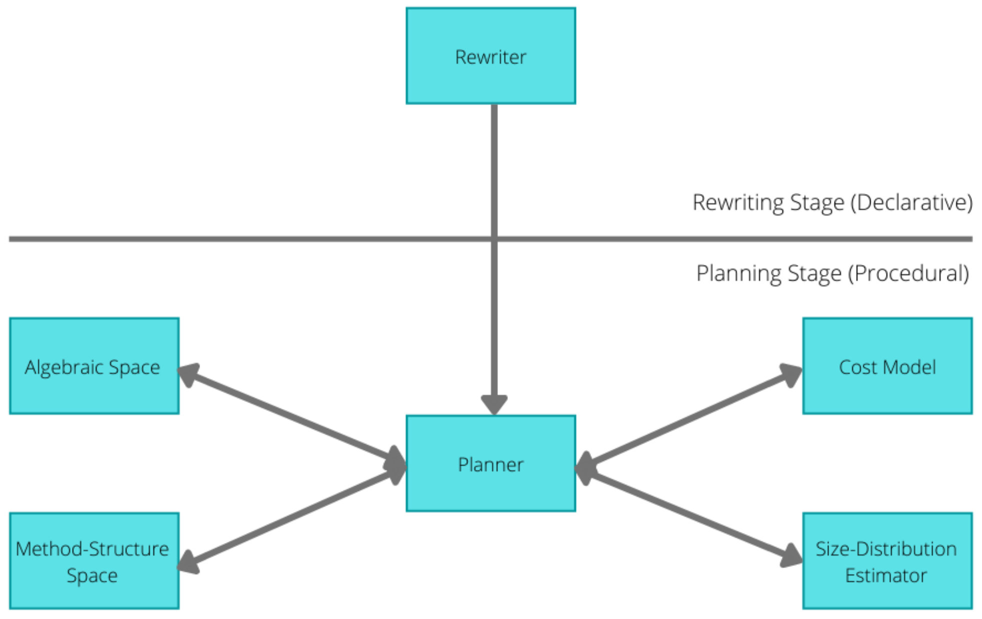
Rewriter
Rewriting Stage (Declarative)
Planning Stage (Procedural)
Algebraic Space
Cost Model
Planner
Method-Structure Space
Size-Distribution Estimator

Figure 3: Query optimizer architecture

The above abstraction optimization process could be divided into the rewriting and planning stages. The functionalities of the modules included in these stages will be thoroughly discussed in the following sections.

2.1.2 Modules Functionalities

Rewriter

This module converts the original query into a more efficient equivalent query by performing transformations depending only on the static characteristics of the query. These transformations include operations such as replacements of views with the corresponding definitions, flattening out of nested queries, etc. Since neither the structure nor the data distribution of the given database is considered during rewriting operations, this module is characterized at the declarative level.

Algebraic Space

This module generates all the alternative series of action execution orders for any given query that the Planner should consider answering the query. All these series produce equivalent results, but there is usually significant fluctuation in performance. Each set of action execution orders is represented as either relational algebraic formulas or in tree forms. The complexity of the examined query determines the number of distinct sets. Several restriction policies are applied to diminish the size of the space needed to be explored, as described below.

Lemma 1

"Selections and projections are processed on the fly and rarely generate intermediate relations. Selections are processed as relations are accessed for the first time. Projections are processed as the results of other operators are generated." [2]

For example, given the below query

> **select** name, property
>
> **from** owners, properties
>
> **where** owners.id = properties.owner_id **and** properties.value > 1M.

three different query trees could be generated, as shown in Figure 4.

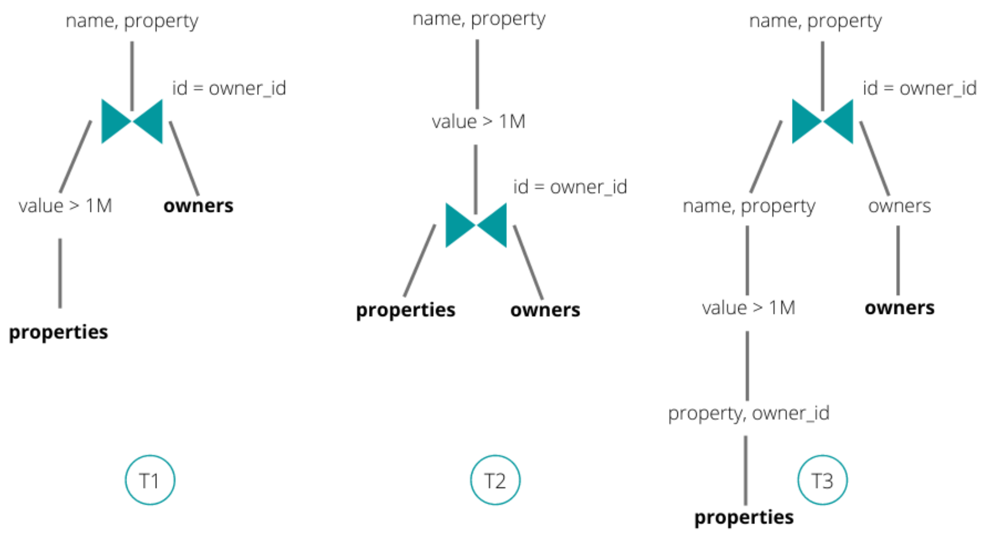

Figure 4: Query trees

Lemma 1 restricts only suboptimal query trees, based on the admission that separate processing of selections and projection incurs additional costs. As a result, only T1 satisfies this restriction: index scan on properties finds only the tuples that satisfy the selection on properties value and joins only those. In contrast, the projection on the result attributes occurs after the join.

Due to join commutativity and associativity algebraic properties, several alternative join series are generated in multi-joint queries. Therefore, a second restriction rule should be applied to reduce the algebraic space further

Lemma 2

"Cross products are never formed unless the query itself asks for them. Relations are always combined through joins in the query." [2]

For example, given the following query

select name, property and area

from owners, properties, areas

where owners.id=properties.owner_id **and** properties.post_code= areas.post_code.

three different query trees could be generated, as shown in Figure 5.

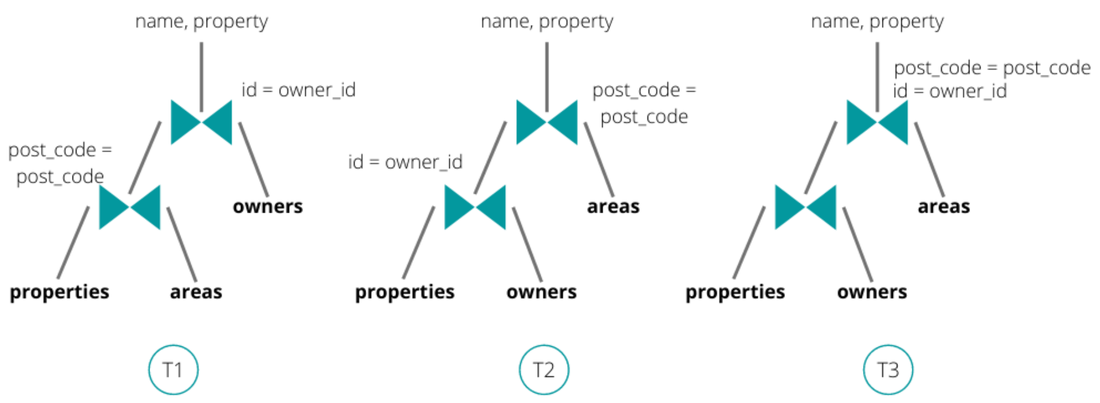

Figure 5: Join trees; T3 has a cross product

This restriction rule eliminates any join trees containing cross products, producing unnecessary large-size results. As a result, T3 is disqualified from the algebraic space.

The final restriction, only present in some database systems such as DB2 and MVS, requiring an even smaller space, deals with the shape of join trees.

Lemma 3

“The inner operand of each join is a database relation, never an intermediate result.” [2]

Given the below query

select name, property, area, state

from owners, properties, areas, states

where owners.id=properties.owner_id **and** properties.post_code= areas.post_code **and** areas.state_id=states.id.

The following cross-product-free join trees can be formed:

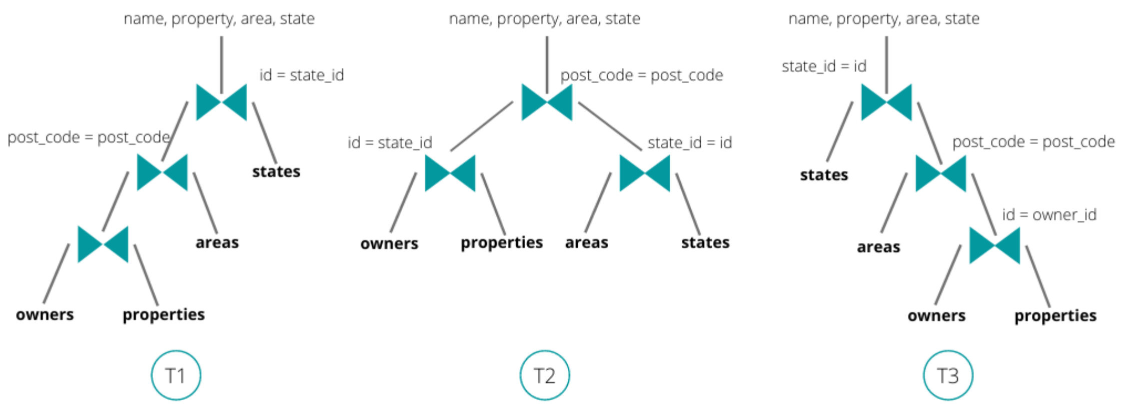

Figure 6: Left-deep (T1), bushy (T2), and right-deep (T3) join trees.

Based on the final restriction, only the T1 join tree is qualified since T2 and T3 include at least one join with an intermediate result as the inner relation. This restriction is more heuristic than the previous ones, and it is possible to eliminate even the optimal plan. However, in most cases, it has been proved that the optimal left-deep is almost equally expensive compared to the optimal tree overall.

This module is classified at the procedural level due to the algorithmic nature of the objects generated during these operations.

Method-Structure Space

Any algebraic expression is composed of a combination of logical operators, such as *(inner/outer) joins, scans, sortings*, etc. Given an algebraic expression or tree from Algebraic Space, this module incorporates any available implementation choice provided by the DBMS. It generates all alternative complete execution plans, specifying the exact implementation of each logical operator included in the execution plan. In other words, method-structure space serves as a one-to-many mapping, matching each logical operator to the available physical implementations. For example, given a join, the number of distinct choices varies based on the available methods used to implement it (e.g., nested loops, merge scan, and hash join) and the indices stored in database catalogs. These alternatives generated at the Method-Structure Space module depend on database structural characteristics and do not affect the development of the query optimizer.

Cost Model

This module evaluates the arithmetic formulas used to estimate the cost of every execution plan within the Algebraic Space with respect to the complexity of the distinct steps that should be accessed to fulfill an execution plan. The cost of each of these steps, including join methods, index type assessments, etc., is determined by simple approximations regarding the underlying functionalities executed by the system during each step. Since these cost formulas depend on assumptions concerning operations like buffer management, disk-CPU overlap, I/O processes, etc., parameters such as the buffer pool size used by the corresponding step, the size of indices and relations incorporated, as well as the distributions of the values on these relations, play a crucial role on step cost estimation.

Size-Distribution Estimator

A critical factor in query optimization is estimating the costs of all available execution plans in advance and without actually invocating them. This can only be achieved by appraising the results of each (sub)query and the frequency distributions of the values in attributes involved in these results. In most cases, a query affects several attributes, and as a result, multi-attribute joint frequency distributions are required to predict the size of the results accurately. However, storing the frequency distributions of all possible attribute combinations in a DBMS is rarely feasible and inefficient. Instead, the attribute value independence assumption is utilized, and even though it is not often true, the joint frequency distribution is calculated as the product of the respective attributes distributions.

Several techniques have been developed to produce accurate estimations over the queries' expected results size and the related attributes' frequency distributions. However, the approach adopted by the most commercial DBMSs involves estimation based on histograms.

A histogram on an attribute X is constructed by partitioning the data distribution of X into β (≥ 1) mutually disjoint subsets called buckets and approximating the frequencies and values in each bucket in some typical fashion [3]. In terms of database systems, the attribute X corresponds to a specific column of a given table. The process of histogram construction involves scanning the database and then aggregating the values fluctuating within a predefined range per attribute. Since the contents of a typical large-scale web resource are frequently updated, the maintenance of the corresponding histograms through a periodic data scan can be proved a highly inefficient task. Instead, adaptive query processing methods can be applied, updating the related histograms on the flight by observing and analyzing the results of the queries that constitute the client-requested workload [4].

Workload-aware self-tuning histograms have been successfully used in relational databases avoiding the costly creation of static histograms of massive datasets. One of the leading and state-of-the-art representatives of the self-tuning approach is STHoles. STHoles' distinct characteristic is allowing buckets to overlap. This more flexible data structure allows STHoles to exploit feedback genuinely multi-dimensional. STHoles allow for inclusion relationships between buckets, resulting in a tree-structured histogram where each node represents a bucket [5]. Holes are subregions of a bucket with different tuple densities and are buckets themselves. A new hole is drilled whenever a query result partially intersects with an existing bucket. The prediction based on the current histogram's statistics diverges from the query results.

Figure 7 shows a bucket b with frequency f(b) = 100. Suppose that from the result stream for a query q. We count those Tb = 90 tuples lie in the part of bucket b that is touched by query q, q ∩ b. We can deduce that bucket b is significantly skewed since 90% of its tuples are located in a small fraction of its volume. The histogram's accuracy will improve if we create a new bucket bn by 'drilling' a hole in b corresponding to the region q ∩ b and adjusting b and bn's frequencies accordingly, as illustrated in Figure 7. So, opening a new hole for the part of the bucket that partially intersects with the query solves the problem of different tuple densities in the same bucket [5].

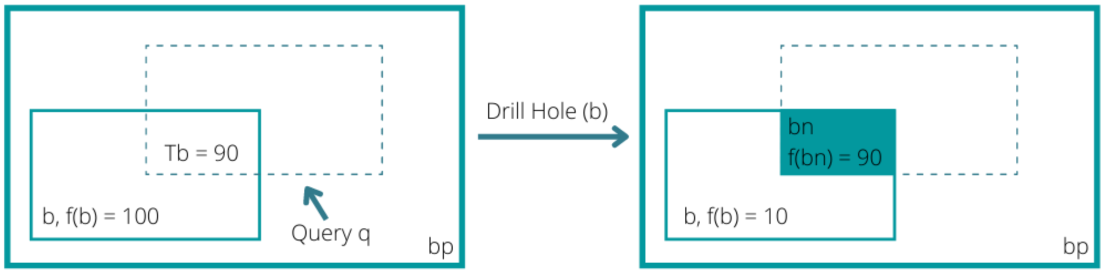

Figure 7: Drilling a hole in bucket b to improve the histogram quality

The number of full buckets stored and maintained is limited due to memory and space resource restrictions. As a result, buckets with relative tuple densities should be merged and replaced with new ones containing more meaningful information. Thus, a penalty function measuring the difference in approximation accuracy between the old and the new histogram is used as a bucket merging criterion. Parent-child merges help eliminate buckets that become too similar to their parents; sibling merges are proper to extrapolate frequency distributions to unseen regions in the data domain and consolidate buckets with similar densities covering nearby regions.

<u>Planner</u>

The core of query optimization occurs in the planner module. The alternative execution plans generated by the Algebraic Space and the Method-Structure Space are filtered. The optimal one is selected based on the established Cost Model with respect to the Size-Distribution Estimator. Several approaches have been proposed based on the exploration strategy employed by the Planner.

Dynamic Programming Algorithms

Search algorithms utilizing dynamic programming strategies are the most commonly used approach in commercial applications. These algorithms can be faced as dynamically pruning exhaustive search algorithms by performing a merge scan on the join trees specified at Algebraic Space and pruning the suboptimal trees that violate the restrictions described at lemmas 1-3 in the previous section. A key component of dynamic programming is the interesting order concept. According to this concept, all join attributes are stored in a sorted queue based on their appearance, starting from the input join relation. Thus, attributes participating in multiple joins can be identified with ease. As a result, it is not acceptable to choose a sub-plan over another, using as criterion just their costs. Instead, their intermediate results should also be considered since the results of the most expensive one may be sorted on an attribute that will save a sort in a subsequent merge-scan execution of a join.

Having stated these, the complete dynamic programming algorithm optimizing a query composed of N relations could be analyzed in the following steps:

Step 1

The input query is processed using a simple sequential scan, identifying all relations alongside the respective partial (single-relation) plans and extracting the exciting order. Afterward, the extracted plans are classified into equivalence classes based on the exciting order. Another class is formed with the plans whose results are not in accordance with the exciting order. Based on the Cost Model, the cheapest plan per in-order equivalence class is selected for further consideration. Finally, the no-order class is scanned, searching for a plan whose cost is lesser than any other plan. Otherwise, the whole class is discarded.

Step 2

The partial plans extracted from step 1 are utilized to generate all possible ways to access every relation joined in the query. These new execution plans are classified and pruned following the same process described in step 1.

…

Step i

Having joined one relation per step, choosing the cheapest plan to access it based on the exciting order, a set of i-1 relations and the individual plans have already been obtained. So, in this step, considering this set of relations plans, it is attempted to join another relation by evaluating all possible ways to achieve it without producing a cross product.

…

Step N

All N relations of the initial query have been joined, and as a result, all possible execution plans have been formed in the previous step. The cheapest plan is selected, marked as the final output of the optimization, and executed to answer the query.

This algorithm avoids enumerating all alternative plans by dynamically pruning those that failed to satisfy restrictions, as described in Lemmas 1-3. As a result, it is guaranteed to determine the optimal execution plan through scanning. In some cases, only O(N^3) plans [2].

Randomized Algorithms

In general, dynamic programming algorithms generate and examine an exponential number of plans to determine the optimal, making the optimization task extremely inefficient. Several algorithms, such as Simulated Annealing, Iterative Improvement, and Two-Phase Optimization, have been recently introduced as an alternative solution to dealing with this dynamic programming inability. These algorithms are based on plan transformations instead of the plan construction of dynamic programming. Specifically, all alternative execution plans are represented as nodes of a graph, each associated with the respective plan's cost. The nodes directly connected to node S are called neighbors of S. A transition from a source node to any destination node is called uphill (resp. downhill) if the latter's cost is higher (resp. lower) than the latter cost of the former. Randomized algorithms perform multiple searches in the graph through random walks (set of moves) to find the globally minimum cost – i.e., reach a node with the lowest cost among all nodes. Some algorithms, Iterative Improvement, achieve optimization by identifying local minimums – i.e., the accepted paths from any given node, allowing uphill moves only after at least one downhill one.

Despite their efficiency over complex queries, randomized algorithms' capabilities are limited due to their strong dependence on the characteristics of the selected cost model and the connectivity of the graphs as determined by the neighbors of each node. Dynamic programming algorithms are generally preferred on simple queries (up to 10 joins) due to their speed and completeness (always find the optimal solution). In contrast, the randomized algorithms are upvoted on more complex queries despite their probabilistic nature due to their efficiency. However, both are affected by the established Cost Model, which depends on optimizer implementation choices and the targeted DBMS data distribution [2].

2.1.3 Distributed Databases

The need to manage and access data stored in distributed data has recently skyrocketed, moving the distributed database system from a small part of the worldwide computing environment a few decades ago to mainstream [6]. The main differences regarding query optimization are detected in the Method-Structure Space and the Planner comparing the centralized case discussed in the previous sections [2]. Additionally, the Cost Model should account for the possible delays due to limitations on the network transmission rates.

Method-Structure Space

Since multiple independent databases are involved regarding Method-Structure Space, additional processing strategies and implementation choices for transmitting data are offered. Additionally, the traditional monolithic execution plans are transformed into a proper combination of web service calls, addressing the distributed databases. However, query processing over distributed web services demands transparent data integration over multiple remote web resources. To this end, the Web Service Management System (WSMS) [7] is utilized as the administration mechanism, enabling the communication and coordination of the individual web services. As a result, a client can query the WSMS, which will handle the optimization and execution of the client's Select-Project-Join query by spanning the multiple connected web services and choosing the optimal ones, and finally returns the corresponding result set Figure 8.

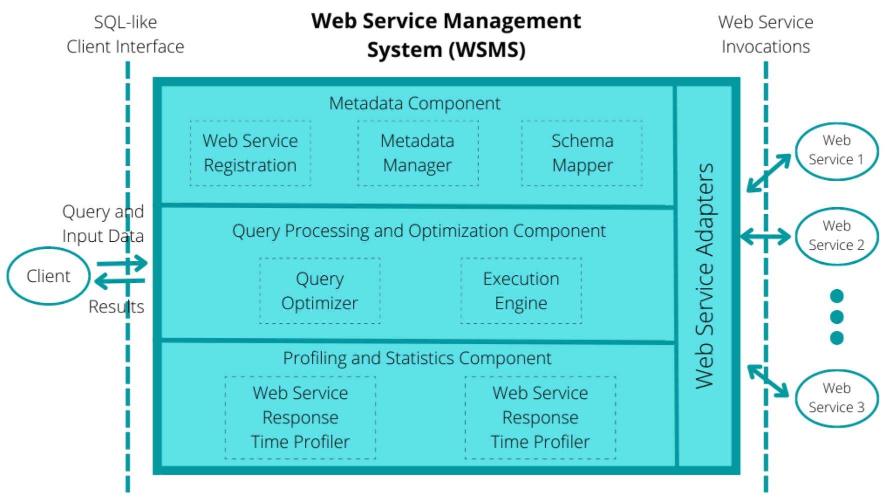

Figure 8: A Web Service Management System (WSMS) [7]

WSMS primarily focuses on constructing the optimal execution plan of available web services that minimize the query's total execution time by exploiting parallelism.

DEFINITION 3.1.1 (SPJ QUERIES OVER WEB SERVICES).

Given a table I, corresponding to the input data by the client and WS1,..., WSn is the set of the available web services, then the class of the queries to be optimized can be described by the following formula:

$$\begin{aligned}&SELECT\ A_s\\&FROM\ I\left(A_I\right) \rhd\lhd WS_1\left(X_1^b, Y_1^f\right) \rhd\lhd \ldots \rhd\lhd WS_n\left(X_n^b, Y_n^f\right)\\&WHERE\ P_1\left(A_1\right) \wedge P_2\left(A_2\right) \wedge \ldots \wedge P_m\left(A_m\right)\end{aligned}$$

Where A_s is the set of projected attributes, A_I is the set of attributes in the input data and P_1, ..., P_m are predicates applied on attributes A_1, ..., A_m, respectively [5].

DEFINITION 3.1.2 (PRECEDENCE CONSTRAINTS).

Suppose a bound attribute in Xj for WSj is obtained from some accessible attribute Yi of WSi. In that case, there exists a precedence constraint $WS_i \prec WS_j$, i.e., in any feasible execution plan for the query, WS_i must precede WS_j [5].

Based on the aforementioned definitions, any query execution plan can be represented as a directed acyclic graph (DAG), whose nodes correspond to the involved web services. If there is a precedence constraint $WS_i \prec WS_j$ between two web services, they will be connected by a directed edge from WS_i to WS_j, implying that the execution should wait for the output of WS_i to invoke WS_j. Otherwise, input data could be dispatched in parallel to the two web services, and thus the corresponding nodes are placed in different paths of the graph, as shown in Figure 9.

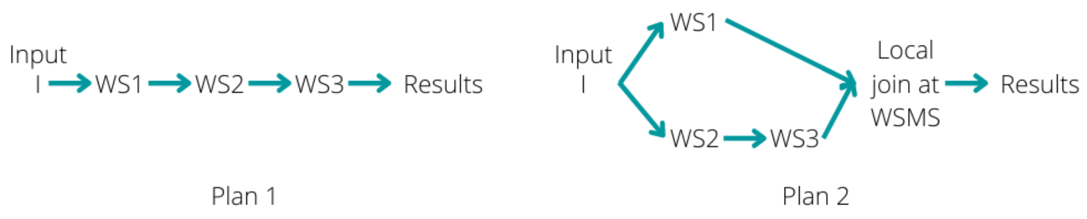

Figure 9: Execution Plan Directed Acyclic Graph (DAG)

<u>Planner</u>

Due to the introduction of parallel and in-order partial plan executions, the need for accurate scheduling escalates the complexity of even simple join queries, as the number of alternative plans is significantly increased. As a result, no dynamic programming algorithm can be applied, making the randomized ones the only feasible solution. However, even optimization algorithms using heuristic methods can be proved relatively inefficient. Scanning a very complex graph while estimating nodes' costs and identifying local or global minima is a time-consuming and expensive process. Additionally, the lack of accurate knowledge of remote web resource data distribution, which can be changed dramatically without any notice, significantly impacts the reliability of cost estimation. More loose methods could be incorporated to overcome these obstacles, replacing the strict analytical processes described in the previous sections. Recent efforts have shown promising results in applying machine learning techniques to query optimization [8]. Such methods will be discussed thoroughly in the following sections.

<u>Cost Model</u>

Establishing a suitable cost function is vital for the query optimization process. The most cases, optimization's primary goal is minimizing query response time. In this project's scope, web services' selectivity and response time will be considered the key factors affecting the overall response time. Based on this assumption, we will focus our efforts on profiling these two quantities to be used for the cost function generation.

Selectivity (s_i)

After evaluating the query and applying all relevant predicates, selectivity si of a web service WSi is a quantity measuring the total number of the returned tuples per input tuple. Since selectivity is a fraction, its value ranges from 0 to 1. For simplicity, in this paper, it is assumed that there is no correlation between web services selectivity.

Response Time (c_i)

Given that $\boldsymbol{r_i}$ is the maximum results invocation rate for a specific web service WS_i. We can define the web service's adequate per-tuple response time as $c_i = 1/r_i$ actually denoting to web service's average invocation cost. This quantity expresses the total time the web service requires to return a result set containing just one tuple. As a result, it incorporates network transmission time, web service processing time, and queuing delays and depends on numerous factors, such as network conditions, web service provisioning, and load. Therefore, it is not acceptable to be considered constant. Instead, a stochastic approach will be adopted, providing a better and more accurate approximation to this quantity.

Cost Function

Assuming that any web service call could be executed in parallel, then the overall system maximum input tuples processing rate will be determined by the web service needed the most time on average per input tuple - i.e., the bottleneck web service. Based on those mentioned above, and assuming that web services selectivities are independent of each other, then the (bottleneck) cost of any DAG, corresponding to an execution plan, will be equal to the maximum of the product of the predecessors' web services combined selectivity and current web service response time, for all web services included in the execution plan [7].

Using the above cost definition, we can calculate the costs corresponding to the query plans shown in Figure 8. Let the respective web services' costs and selectivities be as follows:

i	**1**	**2**	**3**
Cost of WSi (ci)	2	10	5
Selectivity of WSi (si)	0.1	5	0.2

Table 1: Web services' costs and selectivities

and |I| be the number of tuples in input data I. In Plan 1, *WS₁* is the first invoked in the query execution plan, meaning that it has no predecessors and will process the whole |I| number of tuples. Thus, *WS_1* costs equal to $1 \cdot 2 = 2$. and will forward only 10% of them to the following web service, having a 0.1 selectivity value. Thus, *WS_1* cost will equal to $1 \cdot c_1 = 1 \cdot 2 = 2$. Regarding *$WS_2$* the combined selectivity of its predecessors (just *WS_1*) is 0.1, and the respective cost will be $s_1 \cdot c_2 = 0.1 \cdot 10 = 1$. Finally, *$WS_3$* cost value will equal to $(s_1 \cdot s_2) \cdot c_3 = (0.1 \cdot 5) \cdot 5 = 2.5$. So, based on the cost function, the overall cost of the execution plan H is $\max(2, 1, 2.5) = 2.5$.

2.2 Machine Learning Algorithms

Machine Learning (ML) covers a broad range of learning tasks aiming to design and develop computer systems that "automatically improve with experience" while defining the fundamental laws governing all learning processes. ML is a natural outgrowth of the intersection of Computer Science and Statistics. While Computer Science's primary goal is to program computer systems manually, ML focuses on establishing the initial structure that enables a computer system to program itself, using the acquired experience. On the other hand, ML diversifies from Statistics since the former tries to identify the most efficient computational architectures and algorithms that can be incorporated to capture, store, index, retrieve and merge data instead of only extracting conclusions from these data [9].

An abundance of ML algorithms developed so far, organized into a taxonomy based on their desired outcome. This taxonomy involves the following main categories: (i) Supervised learning, (ii) Unsupervised learning, and (iii) Reinforcement learning.

Supervised learning algorithms aim to generate a function mapping inputs to the desired outputs, while unsupervised learning algorithms model a set of inputs without any labeled examples. Reinforcement learning target is establishing a policy guiding an agent to act based on an observation of the world. Every act made by the agent interacts with the environment, which provides feedback used to improve the efficiency of the learning algorithm [9].

During the last decades, various sophisticated learning algorithms have been invented, marking the transition of the artificial neural networks (ANNs) towards increasingly deep neural network architectures with significantly improved learning. However, the complexity of these deep learning models created a set of challenges to overcome due to the induction of black-box properties that can lead to bias and drift in data [19].

2.2.1 Problem Definition

In this project's scope, the primary goal is to predict and reconstruct the optimal execution plan for a given query - i.e., the plan with the minimum cost function, while maintaining the processing time significantly lower than an analytical optimizer. Therefore, the ML algorithm should be trained using a set of samples (training set) corresponding to SQL queries-execution plans pairs produced by an ordinary optimizer for the input set of queries. Given that both the input queries and the optimal execution plan can be considered as text sentences, sequence-to-sequence models [16] can be utilized to translate the input phrase (SQL query) to the respective output (execution plan). As a result, the approach of Neural Machine Translation [17] will be adopted, a state-of-the-art machine translation algorithm, surpassing the typical Recurrent Neural Networks (RNN) and Phrase-Based Machine Translation (PBMT) architectures, providing significantly improved translation speed and accuracy.

2.2.2 Natural Language Processing

Deep neural networks have shown great success in a variety of natural language processing (NLP) tasks, such as language modeling [20], paraphrase detection [21] and word embedding extraction [22], and statistical machine translation (SMT). As stated above, this project's scope will be attempted to face the query optimization problem as a sequence-to-sequence machine translation problem. A natural choice for processing sequential data is the recurrent neural network (RNN).

<u>Recurrent Neural Networks (RNN)</u>

A typical deep neural network assumes that inputs are independent of the outputs. On the contrary, an RNN utilizes information from the previous input to determine the current input and output, introducing a "memory" mechanism. An RNN consists of a hidden state **h** and is fed by a variable-length input sequence **x** = (x_1, x_2, ..., x_n) and generates an output **y**.

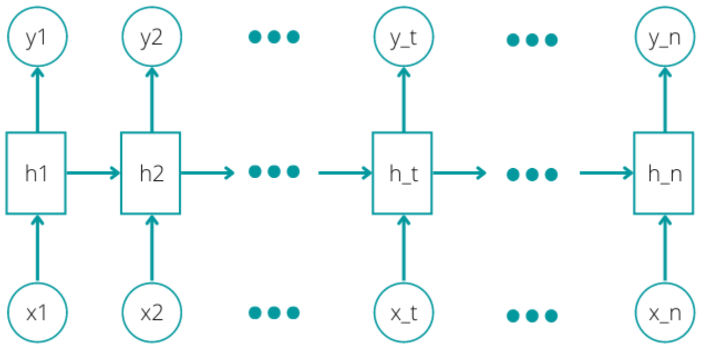

Figure 10: Recurrent Neural Network architecture

As shown in Figure 10, at each step t, the hidden state ht depends on the prior hidden state h_{t-1} and the current input x_t. Since hidden state is a two-factor function, the traditional back-propagation concept is extended to a more complex method called "back-propagation through time". This method unfolds the network in time and calculates each hidden state's gradients with respect to all the network parameters [23]. However, this complex process enhances the vanishing gradient problem, making RNNs' training procedure nearly impossible.

To address the problem of RNNs' long-term dependencies, leading to vanishing gradient during back-propagation, long-short-term memory (LSTM) neural networks come into play [24]. LSTMs' hidden state layer comprises four distinct units controlling the information flow through the network layers: the forget gate that decides what information should be thrown away or kept. The input gate, which determines which values should be updated, the cell state, the information flow bus, and the output gate, which decides what the next hidden state should be, are shown in the following figure.

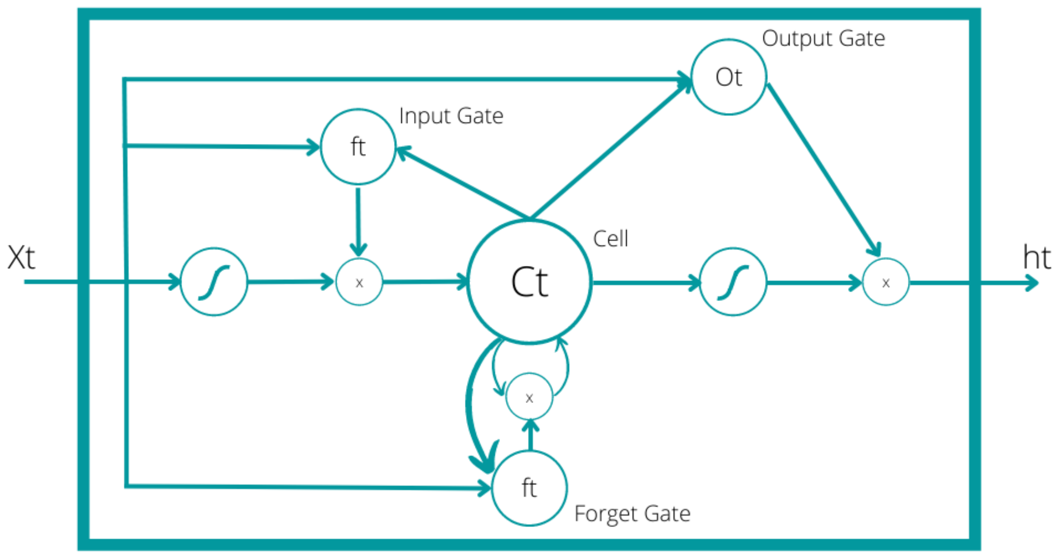

Figure 11: LSTM hidden state unit

2.2.3 Text preprocessing

The instruction of a complete, compact, and meaningful representation for both the input and the output-target sequence is a prerequisite for any RNN-based deep neural network. This representation refers to text vectorization in the natural language processing field, transforming any text sequence into a numeric vector. This procedure could be analyzed into distinct sequential steps: sentence tokenization, vocabulary extraction, sentence transformation, and sentence padding.

Sentence tokenization

The tokenization step's primary task is to split any given sequence into components. Speaking of text sentences, these parts could consist of the individual letters and words or even the collection of sequential letters or words. In the current projects, words are chosen as the unit of the elementary sentence. So, the term token will refer to the word from now on. Before extracting the distinct tokens, it's crucial to perform some cleansing techniques to remove redundant words and symbols and speed up the process. These techniques involve lower case transformation and numbers and symbols removal. Since the processed sentences are SQL statements, SQL operators are critical, and thus they are not removed. Additionally, whitespaces are added to separate the remained tokens. For example, given the previous sentence, the cleansing techniques will perform the following transformations:

Step	Output
initial text	SQL operators such as !, %, <, etc., are not excluded, while the irrelevant symbols like ~, #, etc., are removed.
lower case transformation	sql operators such as !, %, <, etc., are not excluded, while the irrelevant symbols like ~, #, etc., are removed.
remove numbers and symbols	sql operators such as !, %, <, etc., are not excluded, while the irrelevant symbols like , , etc. are removed.
whitespaces addition	sql operators such as ! % , < , etc . are not excluded , while the irrelevant symbols like , , etc . are removed .

Table 2: Text cleansing steps

Having performed the above-described steps, the tokenization process will split the final output text into a list containing the selected components. Single-word tokenization has been chosen for both input and output text sequences in this project's scope.

Token embeddings

Transforming the input text sequences into numeric ones is essential to training any neural network. So, a general transition map needs to be established, matching every token of the initial sequence to a unique numeric representation. This procedure includes extracting the specific words in the space of the sentences' corpus, a.k.a. the *vocabulary*, followed by creating an embedding set, associating each word contained in the vocabulary with a distinct embedding. Several techniques have been introduced regarding the formation of efficient embeddings. This project's scope will examine two strategies: the traditional bag-of-words approach [28] and the word embedding implementation using GloVe word representation [28].

Bag-of-words

The bag-of-words is the most straightforward approach used in natural language processing to create a document representation. Specifically, a list of the unique tokens included in the document is extracted, and each word's frequency (or presence) is calculated per sentence and used as a feature. For example, given the following document consisting of two sentences:

Mary also likes to watch football games. (1)

John likes to watch movies. Mary likes movies too. (2)

the list of the different words (feature vector) is composed of the following words:

"John","likes","to","watch","movies","Mary","too","also","football","games", ".".

So, the sentences will be represented by the following sequence:

[“John”: 0, “likes”: 1, “to”: 1, “watch”: 1, “movies”: 0, “Mary”: 1, “too”: 0, “also”: 1, “football”: 1, “games”: 1, “.”: 1] -> [0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1]

[“John”: 1, “likes”: 2, “to”: 1, “watch”: 1, “movies”: 2, “Mary”: 1, “too”: 1, “also”: 0, “football”: 0, “games”: 0, “.”: 2] -> [1, 2, 1, 1, 2, 1, 1, 0, 0, 0, 2]

However, term frequency is not always a reliable feature since high-frequency words include limited information and thus have predictive power. The term frequency–inverse document frequency (TF-IDF) metric has been introduced to address this issue. TF-IDF is a statistic metric reflecting the word importance over a sentence in a collection [31]. This time instead of term frequency, the TF-IDF score is calculated per word and is used to create the representation of the sentence. As a result, it works as a weighted factor, increasing proportionally to each word in the sentence. At the same time, it is inversely proportional to the number of sentences in the corpus containing the word, which helps to adjust for the fact that some words appear more frequently.

Given the above example, term frequency is calculated per sentence word:

<u>Sentence 1:</u>

$$tf("Mary", s_1) = tf("also", s_1) = tf(\text{"likes"}, s_1) = tf(\text{"to"}, s_1) = tf(\text{"watch"}, s_1)$$
$$= tf(\text{"footbal"}, s_1) = tf(\text{"games"}, s_1) = tf(\text{"."}, s_1) = {}^{1}\!/_{7} = 0.125$$

<u>Sentence 2:</u>

$$tf(\text{"John"}, s_2) = tf(\text{"to"}, s_2) = tf(\text{"watch"}, s_2) = tf(\text{"Mary"}, s_2)$$
$$= tf(\text{"too"}, s_2) = {}^{1}\!/_{11} \approx 0.091$$

$$tf(\text{"likes"}, s_2) = tf(\text{"to"}, s_2) = tf(\text{"movies"}, s_2)$$
$$= tf(\text{"."}, s_2) = {}^{2}\!/_{11} \approx 0.182$$

Afterward, inverse document frequency per word is measured:

$$idf(\text{"John"}, D) = idf(\text{"also"}, D) = idf(\text{"football"}, D) = idf(\text{"games"}, D)$$
$$= idf(\text{"too"}, D) = idf(\text{"movies"}, D) = \log\left({}^{2}\!/_{1}\right) \approx 0.301$$

$$idf(\text{"Mary"}, D) = idf(\text{"likes"}, D) = idf(\text{"to"}, D) = idf(\text{"watch"}, D)$$
$$= idf(\text{"."}, D) = \log\left({}^{2}\!/_{2}\right) = 0$$

Finally, tf-idf scores for sentence one words will be the followings:

$$tfidf(\text{"Mary"}, s_1, \mathrm{D}) = tf\left(\text{"Mary"}, \mathrm{s}_1\right) \times idf\left(\text{"Mary"}, \mathrm{D}\right) = 0.125 \times 0 = 0$$

$$tfidf(\text{"also"}, s_1, \mathrm{D}) = tf\left(\text{"also"}, \mathrm{s}_1\right) \times idf\left(\text{"also"}, \mathrm{D}\right) = 0.125 \times 0.301 \approx 0.038$$

$$tfidf(\text{"likes"}, s_1, \mathrm{D}) = tf\left(\text{"likes"}, \mathrm{s}_1\right) \times idf\left(\text{"likes"}, \mathrm{D}\right) = 0.125 \times 0 = 0$$

$$tfidf(\text{"to"}, s_1, \mathrm{D}) = tf\left(\text{"to"}, \mathrm{s}_1\right) \times idf\left(\text{"to"}, \mathrm{D}\right) = 0.125 \times 0 = 0$$

$$tfidf(\text{"watch"}, s_1, \mathrm{D}) = tf\left(\text{"watch"}, \mathrm{s}_1\right) \times idf\left(\text{"watch"}, \mathrm{D}\right) = 0.125 \times 0 = 0$$

$$tfidf(\text{"football"}, s_1, \mathrm{D}) = tf\left(\text{"football"}, \mathrm{s}_1\right) \times idf\left(\text{"football"}, \mathrm{D}\right) = 0.125 \times 0.301 \approx 0.038$$

$$tfidf(\text{"games"}, s_1, \mathrm{D}) = tf\left(\text{"games"}, \mathrm{s}_1\right) \times idf\left(\text{"games"}, \mathrm{D}\right) = 0.125 \times 0.301 \approx 0.038$$

$$tfidf(\text{"."}, s_1, \mathrm{D}) = tf\left(\text{"."}, \mathrm{s}_1\right) \times idf\left(\text{"."}, \mathrm{D}\right) = 0.125 \times 0 = 0$$

$$tfidf(\text{"John"}, s_1, \mathrm{D}) = tf\left(\text{"John"}, \mathrm{s}_1\right) \times idf\left(\text{"John"}, \mathrm{D}\right) = 0 \times 0.301 = 0$$

$$tfidf(\text{"movies"}, s_1, \mathrm{D}) = tf\left(\text{"movies"}, \mathrm{s}_1\right) \times idf\left(\text{"movies"}, \mathrm{D}\right) = 0 \times 0.301 = 0$$

$$tfidf(\text{"too"}, s_1, \mathrm{D}) = tf\left(\text{"too"}, \mathrm{s}_1\right) \times idf\left(\text{"too"}, \mathrm{D}\right) = 0.125 \times 0.301 \approx 0.038$$

and the representation sequence will be formed as follows:

[“John”: 0, “likes”: 0, “to”: 0, “watch”: 0, “movies”: 0, “Mary”: 0, “too”: 0.038, “also”: 0.038, “football”: 0.038, “games”: 0.038, “.”: 0] ->

[0, 0, 0, 0, 0, 0, 0.038, 0.038, 0.038, 0.038, 0].

Using the same methodology, sentence 2 feature vector will be:

[0.038, 0, 0, 0, 0.038, 0, 0, 0, 0, 0, 0].

Word Embedding

Word embedding is a word representation used for text analysis. Each word is replaced by a fixed-sized and real-valued vector, calculated from the probability distribution for each word to the similar meaning words. As a result, each word representation encloses the context information so that words with similar meanings are expected to be mapped closer to the vector space. Several word embedding models have been developed during the past decade, such as Google's Word2Vec [29], Facebook's FastText [32], Stanford's GloVe [30], etc. Their main difference is in the document corpus used during the training process. In the current project, Sandford's GloVe embeddings implementation has been chosen.

Comparison

Having presented the implementation details regarding these vectorization approaches, it is clear that the BoW algorithm relies solely on contword frequencies under the unrealistic word-independent assumption, while the GloVe embeddings are a more sophisticated method since it encapsulates structural and context information. As far as it concerns the vectorization of SQL queries and the corresponding execution plans, both solutions can be a viable option since creating global representation per word can assist the model in identifying patterns and thus associate the input sequences with the output ones and generate more accurate predictions. A critical difference between these two implementations is the word context information that Word embeddings offer, the positive or negative impact of which should be evaluated since it is not clear if the context of the words in a SQL query (column/table names, SQL keywords) contains meaningful information or adds bias into the system.

2.3 Remarks

To sum up, the typical analytical query optimizers provide accurate optimization results. However, they can be proved highly inefficient when dealing with large and complex queries, joining data from multiple remote databases. Their strong dependence on database data distribution increases their vulnerability to sudden and unnoticed internal changes in the web resources. As a result, the efficiency of analytical optimizers on query optimization tasks over distributed databases can be questionable.

An alternative perspective will be proposed on this project's scope, replacing the analytical optimization approach with a statistical one, incorporating learning techniques. Thus, instead of calculating the exact execution plan through complex tree traversal operations to determine the optimal plan, the proposed model will incorporate natural language processing techniques transforming the original optimization problem into a sequence-to-sequence text generation. As a result, the optimization process is expected to be accelerated and simplified, with a minimum accuracy tradeoff. The following chapters will thoroughly discuss the exact details of the feature extraction, training, and evaluation process to establish this model.

3 Methodology and Experiments

Any traditional deep neural network requires fixed dimensionality inputs and outputs. However, this is impossible for the examined case since neither the input SQL queries nor the output execution plans can have a predefined length. A simple strategy to overcome this challenge is to map the input sequence to a fixed-size vector, using an LSTM network as an encoder, followed by an LSTM decoder mapping this vector to the target sequence [16]. In neural machine translation, this technique was introduced by Google Neural Machine Translation (NTM) systems, replacing the traditional phrase-based translation systems and enabling the capturing of long-range dependencies that occur in natural language sentences. The same long-range dependencies occur in both SQL and query plan statements, making the usage of this architecture quite promising.

3.1 Dataset preparation

The main idea was to deduce the optimization process into a sequence-to-sequence neural machine translator task. The input text sequences will be the SQL queries, and the output text sequence will consist of the optimized execution plan. PostgreSQL database system is utilized to query the database and retrieve the execution plan using the embedded optimizer. A set of queries over several established databases was needed to acquire these input-output tuples. Instead of manually creating the queries and the respective schemas, the CoSQL dataset was used [26]. Create and insert queries from the CoSQL dataset were executed to establish the respective SQL tables in a local PostgreSQL database. Afterward, an *EXPLAIN* command was used for each select query in the dataset to create the corresponding optimal execution plan produced by the PostgreSQL optimizer. For example, given the select query:

```
select t1.first_name from students as t1 join addresses as t2 on          (1
t1.permanent_address_id = t2.address_id where t2.country = 'haiti'         )
```

the optimal execution plan (in JSON format), retrieved by executing the same statement using the *EXPLAIN* command, will be the following:

```
{
  "Node Type":"Hash Join",
  "Join Type":"Inner",
  "Hash Cond":"(t1.permanent_address_id = t2.address_id)"
  "Plans":[
    {
      "Node Type":"Seq Scan",
      "Parent Relationship":"Outer",
      "Relation Name":"students",
      "Alias":"t1"
    },
    {
      "Node Type":"Hash",
      "Parent Relationship":"Inner"
      "Plans":[
        {
          "Node Type":"Seq Scan",
          "Parent Relationship":"Outer",
          "Relation Name":"addresses",
          "Alias":"t2",
          "Filter":"((country)::text = 'haiti'::text)"
        }
      ]
    }
  ]
},
```

corresponding to the following query tree:

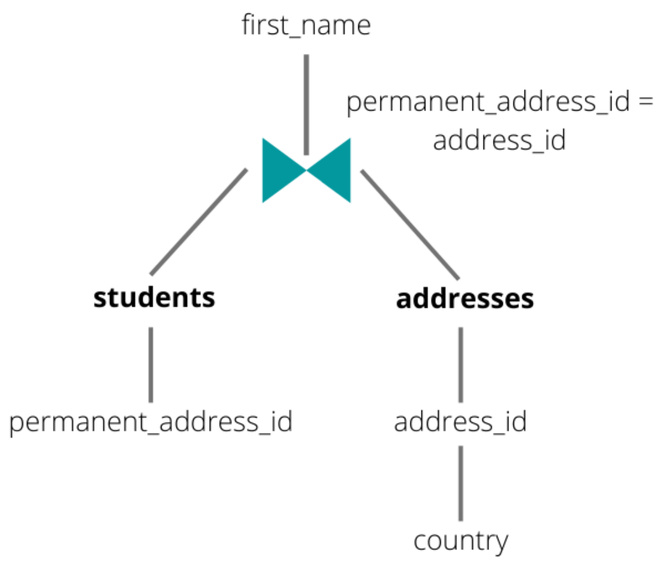

Figure 12: Example query tree

As a result, a set of 5445 queries-execution plans was obtained. This dataset comprises 4784 single-operator queries, 554 queries with two operators, 93 with three, 10 with four, and just four queries with five operators, as depicted in the following graph.

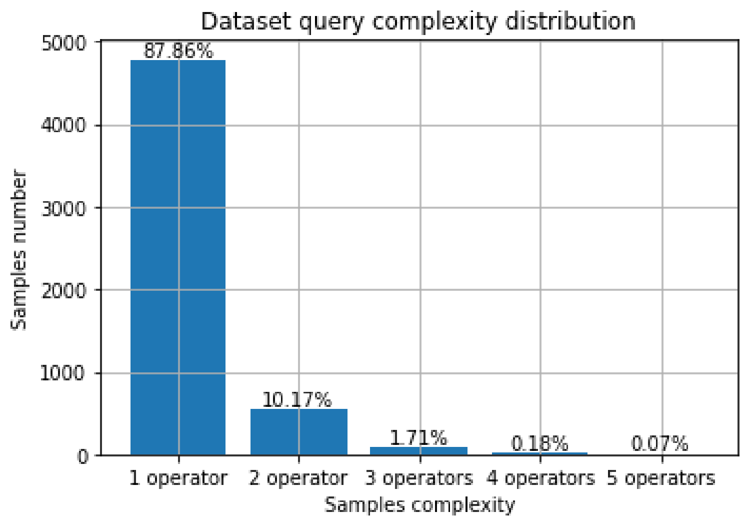

Figure 13: Operator count distribution over dataset queries

This indicates that the acquired dataset consisted of rather than simple queries and can prevent the proposed model from effectively dealing with large and complex queries.

3.2 Data preprocessing

The initial approach did not involve any preprocessing. The plain text from both SQL queries and the respective execution plans was fed into standard text processing pipelines, transforming the raw text into an integer sequence. These techniques will be discussed further in the next section. However, as it can be easily understood, both the input queries and the output JSON format execution plans contain a great deal of redundant information that can mess up the whole training process by increasing model training time, adding bias, and thus plummeting the system's overall efficiency. Therefore, an algorithm transforming input and output texts into a more informative and compact form was introduced.

3.2.1 Input encoding

As far it concerns the query optimization process, the key elements include the contributing tables, the existence of table joins and scans, aggregation and sorting operations, and the projected columns. Our efforts focused on tables' joins and scans in this project's scope, ignoring the aggregation and sorting operations. As a result, the proposed algorithm extracts these elements from the input SQL queries by identifying table aliases, join and scan operations, replacing the joins with the participating table-columns pairs, and the scans with the filtered table columns. For example, table allies are identified given the (1) SQL query mentioned in the previous section. A transition map is created associating table *students* with *t1* alias and table *addresses* with *t2* alias. Afterward, the join operation is spotted and replaced with the following text:

Join students.permanent_address_id-addresses.address_id

where the first word indicates the operation type (join or scan) and the next element corresponds to the join attributes. Similarly, the scan operation is replaced with the following text:

scan addresses.country

To sum up, the initial SQL query text is transformed into the following sequence:

join students.permanent_address_id-addresses.address_id scan addresses.country (2)

The efficiency of this encoding method will be tested compared to the raw input strategy, and the results will be presented in the following sections.

3.2.2 Output encoding

Execution plan encoding

Execution plans preprocessing algorithm is a more complex procedure since PostgreSQL optimizer output is a query tree structure given in JSON format text with multiple nodes. Each element of the query tree corresponds to a specific operation, as described by the node tag *"Node Type"*. Since optimizer output is a query tree, join operations are assigned to tree nodes, while the scan operations are set to tree leaves. Like the input encoding, join nodes are replaced by the join type identifier from the *Node Type* tag, followed by the contributed table-columns pairs. In contrast, the scan nodes are represented by the exact scan type given from the tag *Node Type* and the filtered columns. The critical difference between the two encoders is the operation order. Specifically, the output encoder illustrates the ordering of the operations, which is the optimizer's primary task. Given this, breadth-first traversal is adopted to extract query tree nodes, reassuring that the text representations of the operations lying at a higher level will also come first in the encoded text. For example, given the query (1)

optimizer output will be the following:

Based on the described encoder algorithm, the above query tree will be described by the following text sequence:

inner hash join students.permanent_address_id-addresses.address_id, seq scan students, hash, seq scan, table scan addresses.country (3)

Operators' implementation encoding

Instead of encoding the whole execution plan, it will also be attempted to extract helpful information that an analytical optimizer can use to boost its efficiency. This includes the selection of the optimal physical operator per logical operators described in the original SQL query and determining the actual execution order of the involved operators. On the scope of the current project, we focused just on join and scan operators to reduce the complexity of the encoding procedure. Regarding the physical operators' approach, each logical join operator can be implemented by the query processor module using any of the following procedures:

1. nested loop join
2. merge join
3. hash join

while the scan operators are carried out by:

1. sequential scan
2. index scan
3. index-only scan
4. bitmap (index/heap) scan

So, during the encoding process, using the operators' appearance order in the encoded input sequence, each logical operator will be replaced by the physical implementation described at the corresponding node of the execution plan. As a result, the output encoding procedure discards all intermediate nodes. It extracts only those that enclose the requested information while retaining the one-to-one mapping between the query level operators and the physical ones. For example, the encoded output (see three above) corresponding to the SQL query (presented at 1 of 3.1 section) will be transformed as follows:

hash_join seq_scan (4)

Operators' order encoding

As far as it concerns the operator order at the execution level, an auto-incremented number is added to each operator in the encoded input sequence as a unique identifier to distinguish the first join (or scan) from the following ones. So, the (2) encoded input sequence will be formed as follows:

join_1 students.permanent_address_id-addresses.address_id
scan_1 addresses.country (5)

Afterward, a depth-first postorder traversal in the execution plan tree is conducted. Whenever a node referring to a logical operator is visited, the corresponding unique identifier is added to the output sequence, resulting in a sequence indicating the involving operators' execution order. So, the encoded output sequence will be the following:

```
scan_1 join_1                                                    (6)
```

3.2.3 End-to-end encoding example

To sum up, the encoded input sequence corresponding to the examined SQL query:

```
select t1.first_name from students as t1 join addresses as t2 on
t1.permanent_address_id = t2.address_id where t2.country = 'haiti'
```

will be the following:

```
join students.permanent_address_id-addresses.address_id scan
addresses.country
```

Accordingly, the enriched encoded output, compressing the excessive JSON formatted execution plan of the above query, will be formed as:

```
inner hash join students.permanent_address_id-addresses.address_id,
seq scan students, hash, seq scan, table scan addresses.country
```

while more comprehensive encoded output version, enclosing information about the physical operators' implementation will be the following:

```
hash_join seq_scan
```

Finally, to extract information about physical operators' execution order, the encoded input sequence should be transformed as follows:

select t1.first_name from students as t1 join addresses as t2 on t1.permanent_address_id = t2.address_id where t2.country = 'haiti'

so that the logical operators of the SQL query could be associated with the physical operators of the ordered encoded output:

```
scan_1 join_1
```

3.3 Model

3.3.1 Architecture

As already mentioned, a pair of encoder-decoder is needed to overcome the input-outputs dimensionally variance. A natural choice for sequential data processing and transformation is the RNN. A deep multi-layer unidirectional RNN using LSTM as a recurrent network will be utilized for the input and output decoder in the proposed model. The encoder network transforms the input sentences into a fixed-sized vector at a higher level. In contrast, the decoder network consumes the vector fed by the encoder and decodes the predicted sequence to compare it with the expected one [25].

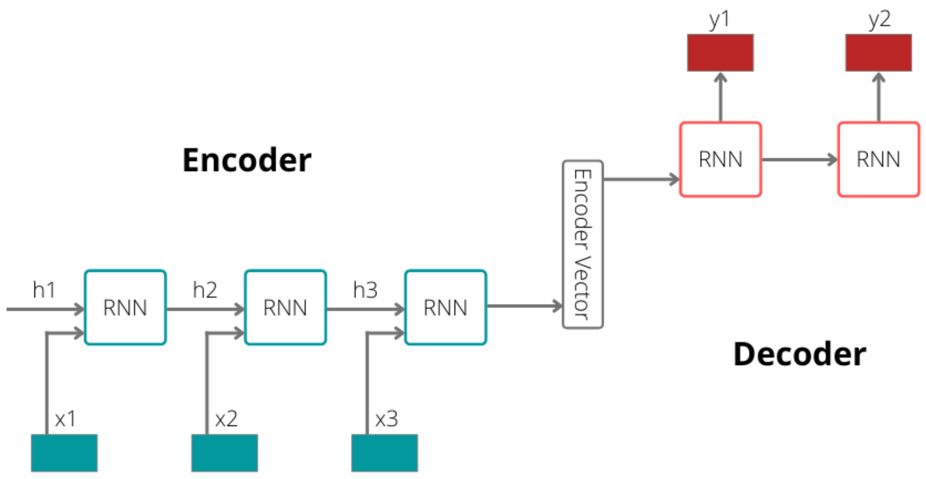

Figure 14: Encoder-decoder sequence to sequence model

As shown in Figure 14, the above-described architecture's main disadvantage is that the information from the first tokens in the input sequence is diluted, especially in long sequences, due to encoder output vector size restrictions. To address this issue, a pioneering technique called *Attention Mechanism* was introduced by Bahdanau et al. (2014) [33] and Luong et al. (2015) [34]. The attention mechanism improves system efficiency by allowing the decoder to access all the past encoder's hidden states and emphasize the most relevant ones. The measure of each of the encoder's past state importance to the decoder output is denoted by the *alignment vector*. The alignment vector has the same length as the input sequence. It is calculated at every time step of the decoder using the concat strategy, which involves the addition of the decoder hidden state and the encoder hidden states, followed by a linear layer with a tanh activation function and multiplied by a weight matrix. Thus, each of its values reflects the importance of the corresponding word in the source sequence.

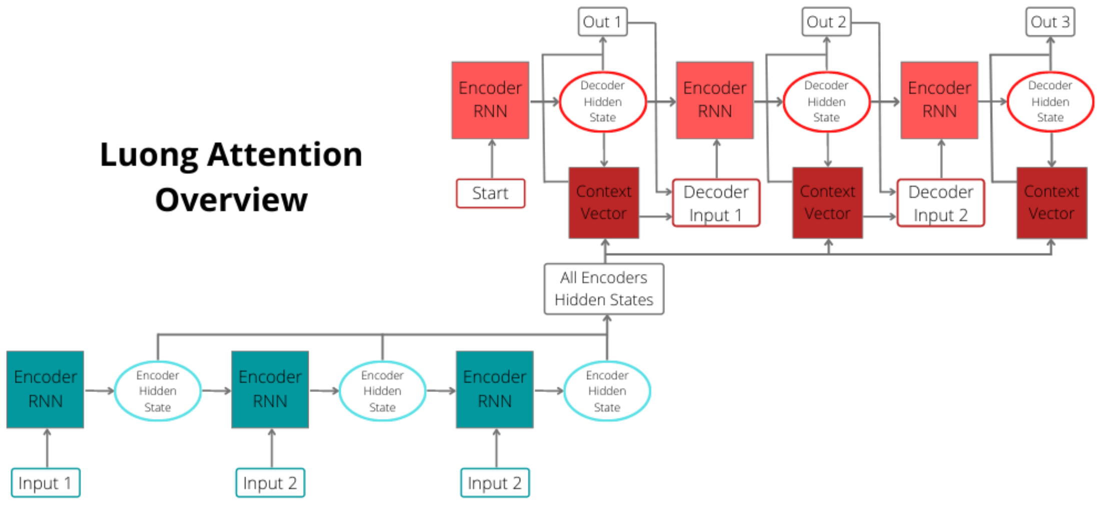

Figure 15: Attention Mechanism

3.3.2 Training process

State-of-the-art recurrent neural networks in the field of Natural Language Processing use *Teacher Forcing* [35] algorithm in the training process. Teacher forcing key characteristic is that it trains recurrent networks by supplying the actual output sequence values as the next timestep's inputs improving the network's learning capabilities using multi-step sampling.

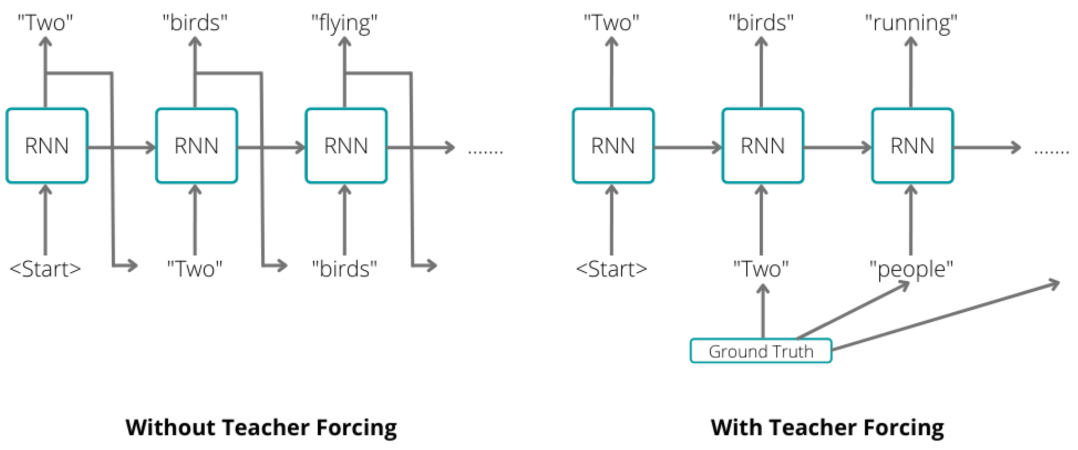

Figure 16: Teacher Forcing mechanism

In the proposed model, this technique is applied by providing the target sequence at timestep t to the decoder input at time step t+1. To achieve this time-shifting, a start token,

<sos> is added as the leading element of each input sequence, while an end token, <eos> is inserted as the target sequence trailing element, equalizing the sequences' lengths and denoting their end.

3.3.3 Inference procedure

The inference procedure involves the generation of predictions given a source sequence. The source sequence is fed to the encoder to produce the encoder's hidden state, used to initialize the decoder. Afterward, <sos> token is supplied to the decoder, which produces an output per time step. The decoder's output is handled as a set of logits corresponding to a word. The word associated with the maximum logit value is the timestep's output. The prediction process is terminated when the end token, <eos> is generated.

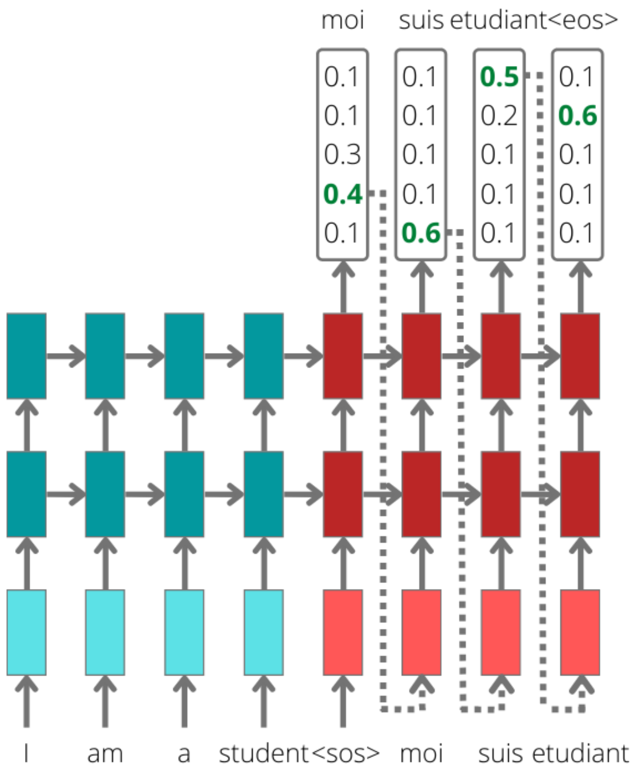

Figure 17: Inference procedure

3.4 Experiments

3.4.1 Glove Embeddings experiments

In the following set of experiments, the GloVe 200-dimensional word representation and Word2vec algorithm were utilized during the vectorization of the input and output text sequences.

Raw data experiment

As mentioned in the previous section, the initial thought was to develop a model that could reconstruct the complete execution plan using the simple SQL query as input. Therefore, the input dataset was composed by appending the SQL queries of the CoSQL dataset. At the same time, the output-target sequences were created by the corresponding JSON-format output of the EXPLAIN SQL command, i.e., the optimal execution plan. As a result, a total of 5445 query-plan pairs were created. Then, the Word2Vec library and GloVe word embeddings, with the word representation vector size of 128 elements, were utilized for training two separate embedding models-encoders: an input encoder and an output encoder. The embedding vectors of the former were created using the input dataset, while the latter's vectors were based on the output dataset. The input encoder transformed, whose vocabulary consisted of 1151 unique tokens (words), the SQL queries into numeric vectors, while the output encoder, with a 1935-token vocabulary, vectorized the execution plans. Padding was applied in both input and output vectorized sequences to equalize their length. After the padding sequence, the input dataset comprises 119-element vectors, while the output dataset is 1385-element vectors.

As a result, 5445 fixed-sized input-target pairs were created and split into the train, validation, and test subsets using the ratio 70:10:20, respectively. The validation dataset was incorporated as an indicator of the early-stopping mechanism with five epochs patient, applied during the model training process, preventing model overfitting.

Regarding the overall model architecture, since the encoder embedding layer dimension should be following the input sequences vocabulary size of 1153 elements, to be able to encode every word in the input corpus, the former dimension was set to 1154. The extra dimension serves as a placeholder for the padding characters. Accordingly, the decoder embedding layer was set to 1936, whereas the output dense layer dimension numbers 1935 elements to recreate any execution plan. Since the pre-trained Glove vector embeddings were already fit to the corpus using the transfer learning utilities of the Word2vec library, the encoder and decoder embedding layers were not trainable during the overall model training process.

Having stated these, the training process is ready to start. However, this task has been proven to be highly inefficient and time-consuming since the excess of both input and output sequences length demands training through multiple timesteps. Specifically, a memory allocation exception was raised during just the first batch of the first epoch, making the model's training process impossible under the available resources (32GB RAM). Therefore, the need to shrink both the input and output sequences length has been proved eminent.

Input-Output encoding experiment

The former experiment highlighted reducing both the input and output sequences. So, the 5445 input-output pairs used in the initial model training process were transformed using the encoding algorithms described in section 3.2. Specifically, the input encoding followed by the Glove embedding encoder was applied to the set of simple SQL queries, resulting in a new input dataset with half the initial vocabulary (559 distinct tokens). Accordingly, the output sequences dataset was also compressed using the execution plan encoding algorithm. As a result, the reduced output dataset contained a total number of 839 unique words. After vectorization and padding, the final dataset consisted of 5445 pairs of 32-token input and 92-token output vectors. The same train, validation, and test split ratio, as well as early-stopping mechanism, was incorporated in this experiment, too. However, since both the input and output sequences length were modified, the dimensions of the encoder and decoder units were fixed accordingly. The encoder embedding layer dimension was 560, while the decoder embeddings size numbered 840 elements, and the dense output layer dimension equaled 840.

As shown in the following graphs, the training process lasted six epochs, and despite the apparent improvement compared with the previous ultimately failed attempt, the results are not satisfying. Neither Loss nor Accuracy curves are smooth and converge, while the accuracy score remains substantially low.

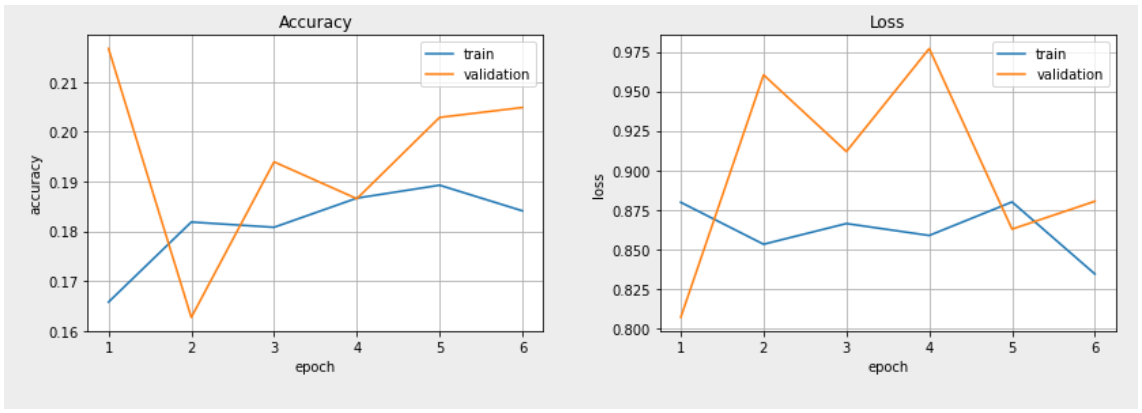

Figure 18: Training-validation accuracy and loss per epoch

Additionally, the testing accuracy score of 30.04% insists that the model's inference capabilities are irrelevant, compared to the 22.66% accuracy score of a dummy baseline model that generates predictions by repeating the most common token. In other words, the predicted sequences are consisted of random words, enclosing no meaningful information. The model's poor performance is due to the design and the overall methodology of this experiment. As a result, the model failed to identify underlying patterns associating input and output sequences. This means that despite the noticeable shrink achieved through encoding algorithms applied to input and output, their complexity remained too high, exceeding the model's capabilities.

Therefore, the task of recreating the whole or partial execution plan based just on information extracted from the initial SQL query cannot be fulfilled using the proposed NTM model. After all, several well-established analytical implementations have already been developed to deal with this problem.

Operator implementation experiment

Having abandoned the efforts to predict the execution plan using as input the SQL query due to the efficiency reasons described above, our focus shifted towards introducing a model that can provide helpful information to the analytical query optimization process, thus enhancing its efficiency.

As mentioned in chapter 2.1.2, a substantial element of the success of the query optimization process is selecting the appropriate physical implementation for each logical operator in the initial SQL query. So, the encoded input sequences used in the previous experiment will be retained during this experimental setup. In contrast, the output sequences will be further compressed by applying the operators' implementation encoding methodology described in section 3.2.2. Further input sequences abstraction is avoided since, based on query optimization theory stated in section 2.1.2, the selection of the optimal physical operator depends on database structural characteristics and predicates distribution, which can be reflected on the tables and columns defined at the query and are already present even in the encoded sequences. Thus, 5445 vectorized input sequences, with a fixed size of 32 elements, will be fed to the encoder module, just as in the previous experiment. However, the feature that distinguishes this setup from its ancestors is that the output sequences are now constituted of vectors whose length equals just six tokens. At the same time, the respective vocabulary contains nine different words, i.e., an enormous 99% size reduction compared to the previous experiment. Subsequently, the overall NTM model was simplified since the number of the encoder's trainable weights plummeted drastically.

These more comprehensive output sequences lead to a significant decline in the training time (from 35 minutes per epoch to less than 1) and a simultaneous improvement in the model's prediction accuracy, as shown in Figure 19.

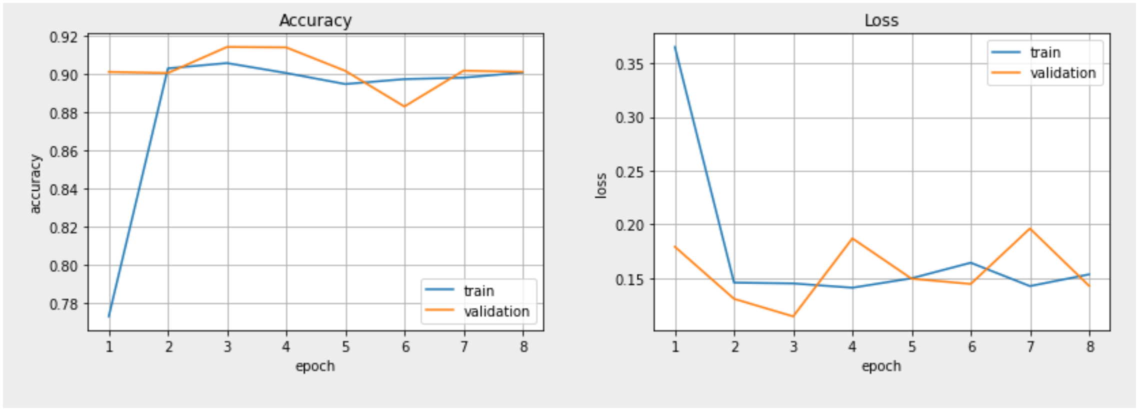

Figure 19: Training-validation accuracy and loss per epoch

Due to the activation of the early-stopping mechanism, the training process terminated at the 8th epoch, with the final model's prediction accuracy equal to a promising 91.62% over 554 test samples. Examining the model efficiency with respect to the queries' complexity, the model creates almost perfect predictions for the most straightforward queries. However, its accuracy diminishes when more complex and multi-operator queries occur.

To demonstrate the objective model's performance, the prediction results are compared with the respective ones generated by a dummy baseline model, whose inferences are just replicates of the most common word in the test dataset. The comparison results are depicted in the following graph:

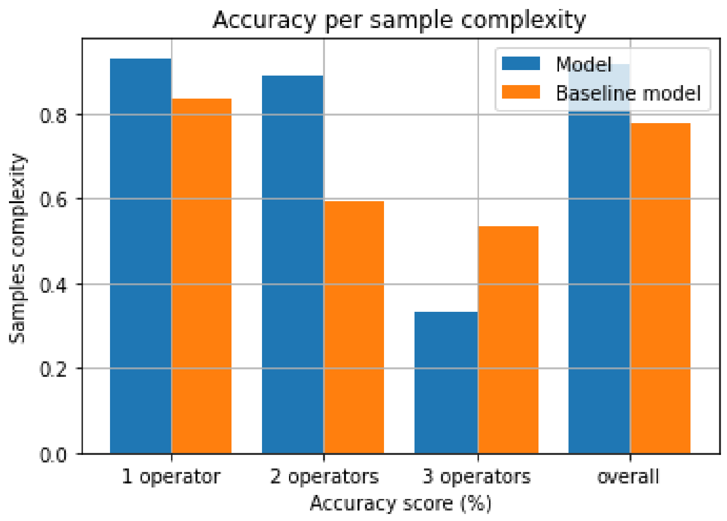

Figure 20: Prediction accuracy compared with query complexity graph

The observed ineffectiveness is expected and caused by the absence of multi-argument queries in the initial dataset. Thus it can be treated by enriching the training dataset with more complex queries.

Operator order experiment

As mentioned in section 2.1.2, several alternative series of operators' execution orders can be generated for any query, producing equivalent results. However, significant fluctuation in the query answering performance can be observed between the different execution series. Therefore, determining the operators' optimal execution order is another vital decision that should be made during the query optimization with a massive impact on the whole process efficiency. In the scope of this experiment, capitalizing on the order output encoding algorithm, we developed a model that determines the execution order of the physical operators participating in the initial SQL query. Since the present model aims to predict the optimal arrangement of the involved operators at the execution step, single-operator queries should be excluded from the training process. However, having rejected these queries from the training corpus, the number of samples declined to only 661 queries. This is undoubtedly a small dataset to train a complex deep learning model. To overcome this issue, more than 100 multi-operator queries were written and added manually to the training dataset. So, the final dataset consists of 796 samples.

Before proceeding with the model training, as stated in paragraph 3.2.2, input sequences should also be transformed by adding a unique identifier to each logical operator to match its physical implementation in the predicted execution order sequence. So, having applied both input and output encodings to the corresponding sample pairs, the input sequences vocabulary numbers 638 unique words, while the output sequences vocabulary included just eight different words. Subsequently, after the vectorization and padding process, the input vector size equals 32, and the output-target vectors are composed of 6 elements. So, the encoder module embedding layer consists of 640 elements, whereas the decoder embedding layer size was set to 9, and the output layer was composed of 9 tokens.

The model training process terminated after ten epochs, giving mediocre results, as expected, due to the fixed training dataset size. The training loss and accuracy curves are presented in the following figures:

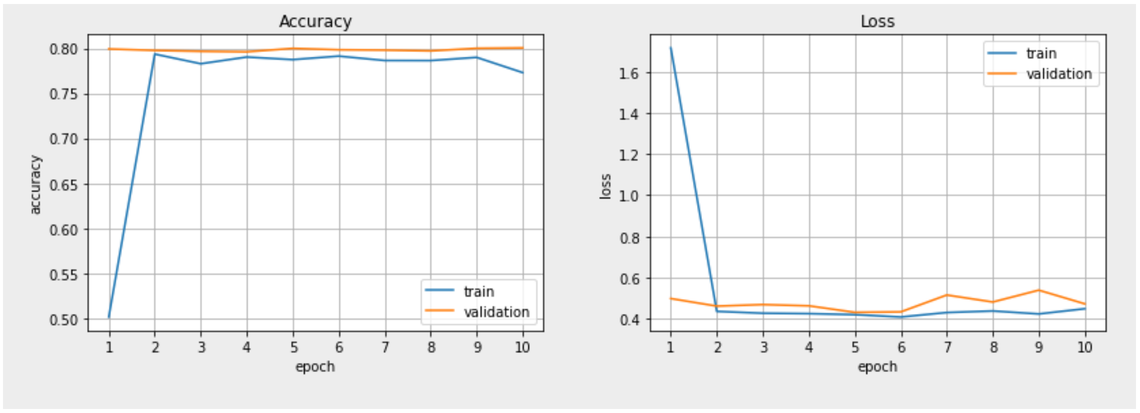

Figure 21: Training-validation accuracy and loss per epoch

The model's prediction accuracy score is confined to 71.77%. In contrast, the inverse correlation between the prediction accuracy and query complexity observed in the previous experiment is retained, too, as shown in the graphs below.

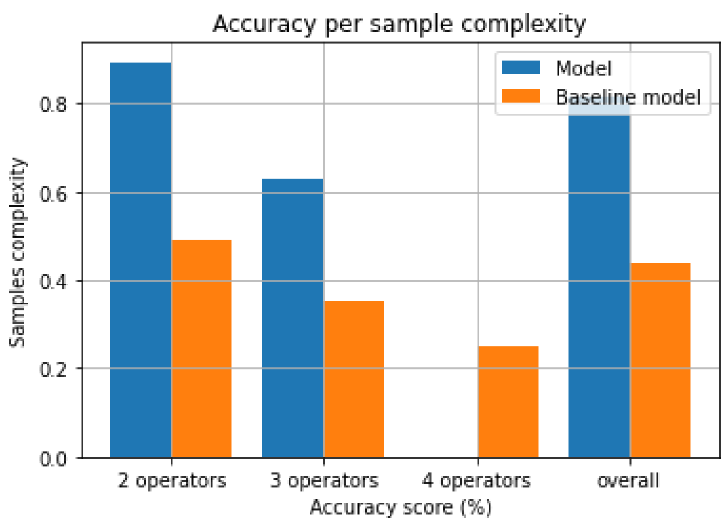

Figure 22: Prediction accuracy compared with query complexity graph

In this case, the baseline model predictions are obtained by replicating the first token of the test-expected text sequence. The above figure provides strong evidence to the previously stated allegation that the model's intermediate results are primarily due to the lack of a sufficient training dataset. The model's prediction results are improved for the most uncomplicated queries with more samples.

3.4.2 BoW experiments

The BoW vectorization approach was applied in the input and output encoded sequences. The last two experiments (i.e., operator implementation and order experiments) were repeated to examine the impact of the vectorization process on the most promising models' efficiency.

Operator implementation experiment

On the scope of this experiment, the same input-output pairs were incorporated as in the operator implementation setup described in the previous section. However, the Bag-of-words approach was utilized instead of the pre-trained GloVe embeddings to vectorize the training samples. Specifically, the unique words were extracted from input and output corpora, formatting the corresponding vocabularies. Afterward, an auto-incremented unique integer was assigned to each word, starting from 1, since the 0 was used as a placeholder for the padding characters. This one-to-one mapping, combined with zero padding, was applied to transform the text sequences into fixed-sized numeric vectors.

Regarding overall model architecture, encoder and decoder modules and their components remained unchanged, with just one key difference. The encoder and decoder embeddings layers variables were not excluded from the training process. In other words, during the back-propagation step, the applied weights modifications affected embeddings variables, too.

After the completion of the training process, which lasted 16 epochs, the final model produces almost identical results to the corresponding experiment with the GloVe embeddings, as shown in the following figure:

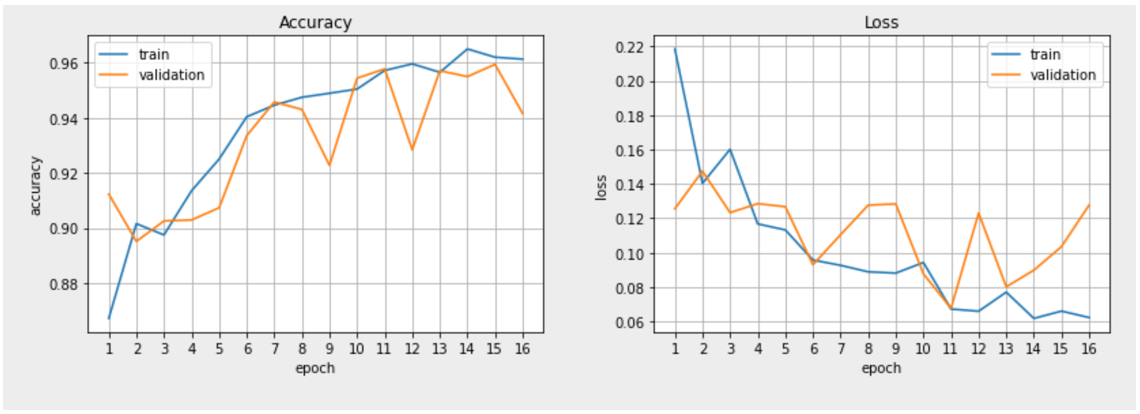

Figure 23: Training-validation accuracy and loss per epoch

The model's inference capabilities are sufficient, giving an overall prediction accuracy score of 92.29%. As expected, the accuracy is downfalls as the queries' complexity rise.

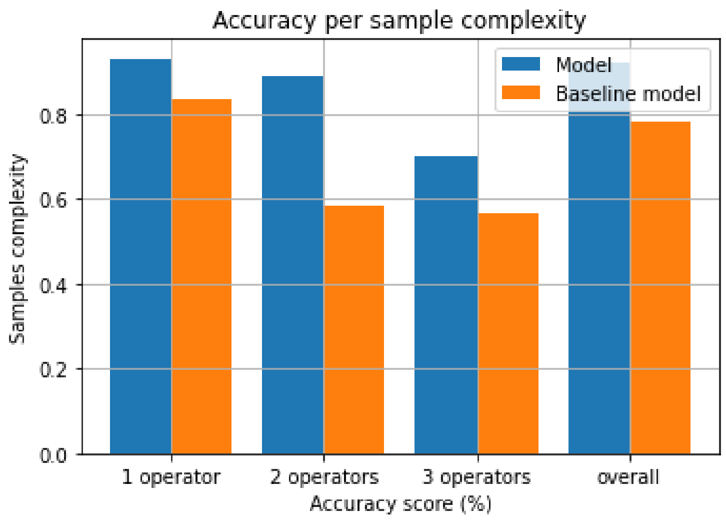

Figure 24: Prediction accuracy compared with query complexity graph

The outcomes similarity between GloVe embeddings and BoW approaches is due to the training dataset's structural characteristics. Notably, the sparsity of both input and output corpora is limited due to the encoding procedures applied. As a result, there are very few rare words, and thus there is a sufficient number of training samples to establish efficient embeddings per word. Additionally, the snake case column naming convention, a common practice in most SQL queries, adds to the training corpora unknown words for the GloVe embeddings. This practically means that the large datasets incorporated during the GloVe representation training process make no difference regarding the examined case study, making their usage almost irrelevant.

The only noticeable advantage of the GloVe embeddings over the BoW is the reduced training time since the number of the network trainable parameters is reduced due to the encoder and decoder's non-trainable embeddings layers – 90.240 fewer trainable parameters involved in the GloVe approach networks, translated to a one-second difference in training time per epoch and about 5.5 minutes (339 seconds) in the whole training process. This feature also gives values to the scalability of the model, considering that in more complicated schemas, with much more extensive input and output vocabularies and thus more massive embedding layers, the training time, as well as the processing resources, could make (just as the first experiment) the whole training process impossible.

Operator order experiment

Having applied the same methodology as described in the previous paragraph, concerning the vectorization algorithm, on operator order encoded input-output sequences produce results that confirm the above-stated allegations. Specifically, BoW implementation slightly outperforms the GloVe embeddings approach regarding the models' prediction accuracy. At the same time, the latter provides improved training speed and thus a more promising perspective in more extensive and more complex training tasks, as depicted in the subsequent figures.

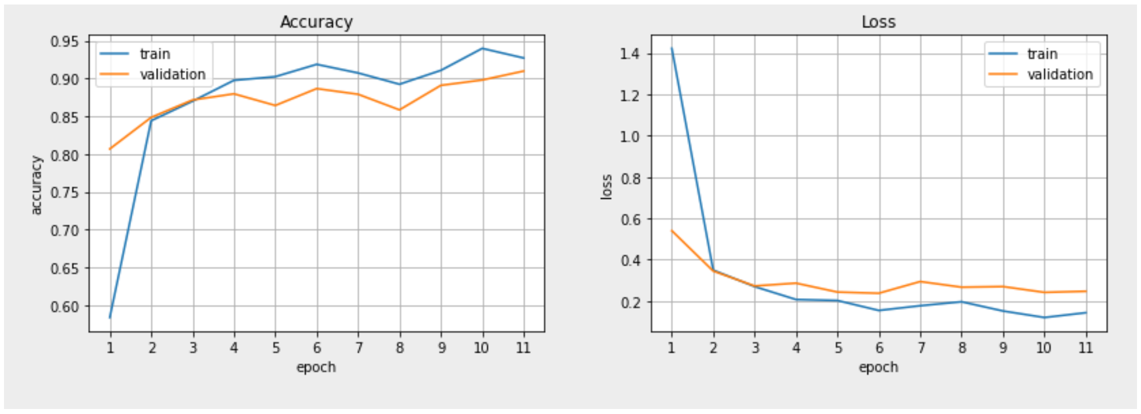

Figure 25: Training-validation accuracy and loss per epoch

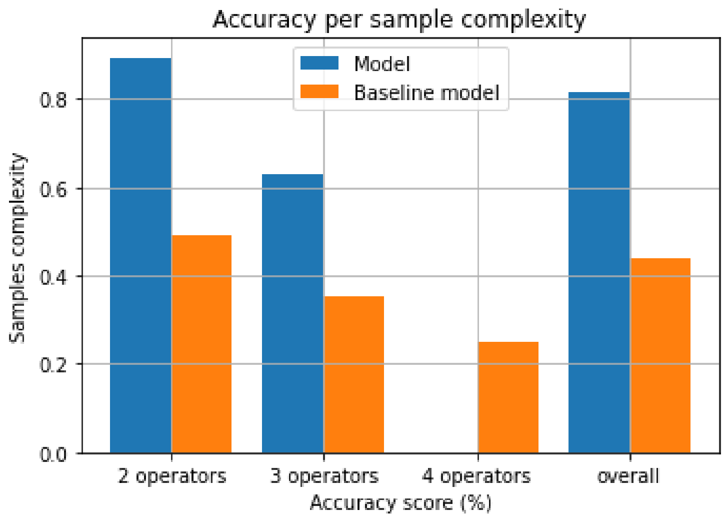

Figure 26: Prediction accuracy compared with query complexity graph

3.4.3 Discussion

After conducting these experiments, it is clear that the task of the complete execution plan recreation, using the respective SQL query as input, cannot be fulfilled using the natural language processing techniques NTM architecture under the available hardware resources. On the other hand, the extraction of valuable insights concerning minor decisions should be made during the optimization process, such as the selection of the suitable implementation per logical operator participating in the original SQL query, or even the optimal operators' execution order, is a pretty promising perspective, that can accelerate the whole process.

Regarding the NLP-related task of studying the impact of the vectorization algorithm on the final model's inference efficiency, the experimental results insist that the old-fashioned Bag-of-Words approach leads to more accurate models due to the specific characteristics of this problem (columns-tables name, SQL keywords, etc.). On the other hand, the GloVe embeddings provide a more versatile and scalable solution that can make a strong case in datasets containing more complex queries.

In conclusion, the dataset's structure used to train the models was the primary vulnerability of the whole task, diminishing the proposed model's efficiency and the generalization of the drawn conclusions. Specifically, the lack of a sufficient number of complex queries limited the model's performance in more complicated optimization tasks. It reduced the confidence level of its effectiveness in real-world applications.

4 Conclusion

In this project's scope, natural language processing techniques combined with Neural Machine Translation architectures were incorporated to reconstruct the optimal execution plan of any given SQL query by replacing the analytical approach used by the typical optimizers with probabilistic deep learning procedures. However, the complexity and the sparsity of both the raw and the encoded and compressed input and output sequences exceed the learning capabilities of the proposed deep neural network, producing inefficient or even non-trainable (resource-wise) models.

Having abandoned the execution plan prediction task due to the reasons already mentioned, we focused on extracting valuable insights that the ordinary optimizers can use as hints to conclude faster and more accurate decisions during the optimization process regarding operators' implementation and execution order. Encoding algorithms were introduced to filter the initial input and output texts and extract the most meaningful words, including relevant information. For simplicity and importance reasons, only scan and join operations were considered. Both models show promising results. Although, primarily due to the limited complex queries involving multiple operators included in the training dataset, the models' efficiency drops as the query complexity rises.

Finally, the effect of text sequence vectorization techniques on the models' efficiency was also examined. Specifically, the GloVe embeddings and Bag-of-words approaches were validated for the operator implementation and the execution order models. Having applied both of these algorithms in the training dataset, it was found that the former provides better convergence time during the training process and improved scalability, whereas the latter result in models with slightly enhanced inference capabilities.

o Future Work

The limited amount of training samples, especially those involving complex queries, hugely impacted the proposed models' efficiency during the whole process. To overcome this obstacle, the training dataset should be enriched with more multi-operator queries. Since this process could be proved to be a demanding and time-consuming task, another perspective that can address this issue is the integration of sample-weighting architectures so that the limited complicated sample has a more significant impact on the training process.

Another improvement field involves including the rest of the SQL operators cut off in this project's scope. Additionally, embedded select statements were also excluded from the encoding algorithms and should be considered in the future.

Finally, ways of incorporating the extracted information regarding the operator implementation and execution order in the optimization process should also be studied as a real-world evaluation of the present work findings.

References

[1] S. Chaudhuri, An Overview of Query Optimization in Relational Systems, PODS '98, 1998.

[2] Yannis E. Ioannidis, Query Optimization. ACM Computing Surveys, Volume 28, Issue 1, pages 121–123, 1996.

[3] Yannis Ioannidis, The History of Histograms (abridged), Proceedings 2003 VLDB Conference, pages 19-30, 2003.

[4] Katerina Zamani, Angelos Charalambidis, Stasinos Konstantopoulos, Nickolas Zoulis, and Effrosyni Mavroudi. Workload-Aware Self-tuning Histograms for the Semantic Web. In Transactions on Large-Scale Data-and Knowledge-Centered Systems XXVIII, pages 133–156. Springer, 2016.

[5] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. STHoles: A Multi-dimensional Workload-Aware Histogram. In ACM SIGMOD Record, Volume 30, pages 211–222. ACM, 2001.

[6] Özsu M.T., Valduriez P. (2020) Introduction. In: Principles of Distributed Database Systems, pages 1-2. Springer, Cham.

[7] Utkarsh Srivastava, Kamesh Munagala, Jennifer Widom and Rajeev Motwani. Query Optimization over Web Services.

[8] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In Proceedings of the 2021 International Conference on Management of Data. Association for Computing Machinery, New York, NY, USA, 1275–1288.

[9] Taiwo Oladipupo Ayodele, Types of Machine Learning Algorithms. In New Advances in Machine Learning, pages 19-48, February 2010.

[10] Sebastian Ruder, An overview of gradient descent optimization algorithms, June 2017.

[11] Evgeniou T., Pontil M. (2001) Support Vector Machines: Theory and Applications. In: Paliouras G., Karkaletsis V., Spyropoulos C.D. (eds) Machine Learning and Its Applications. ACAI 1999. Lecture Notes in Computer Science, vol 2049.

[12] D. Florescu, A. Levy, I. Manolescu, and D. Suciu. Query optimization in the presence of limited access patterns. In Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data, pages 311–322, 1999.

[13] H. Garcia-Molina et al. The TSIMMIS approach to mediation: Data models and languages. Journal of Intelligent Information Systems, 8(2):117–132, 1997.

[14] Tom M. Mitchell, The Discipline of Machine Learning. CMU-ML-06-108, July 2006.

[15] An Overview of Query Optimization in Relational Systems

[16] Ilya Sutskever, Oriol Vinyals, Quoc V. Le, Sequence to Sequence Learning with Neural Networks, 2014

[17] Quoc V. Le et al., A Neural Network for Machine Translation, at Production Scale, 2016

[18] Kyunghyun Cho et al., Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation, 2014.

[19] P. P. Shinde and S. Shah, "A Review of Machine Learning and Deep Learning Applications," *2018 Fourth International Conference on Computing* Communication Control and Automation (ICCUBEA), 2018, pp. 1-6.

[20] Y. Bengio et al., A neural probabilistic language model, 2003.

[21] Socher et al., Semi-supervised recursive autoencoders for predicting sentiment distributions, 2011.

[22] Mikolov et al., Efficient estimation of word representations in vector space, 2013.

[23] P. Werbos. Back-propagation through time: what it does and how to do it. Proceedings of IEEE, 1990.

[24] S. Hochreiter et al., Long Short-term Memory. Neural Computation 9(8):1735-1780, 1997.

[25] LUONG, Minh-Thang. Neural machine translation. 2016. Ph.D. Thesis. Stanford University.

[26] Yu et al., CoSQL: A Conversational Text-to-SQL Challenge Towards Cross-Domain Natural Language Interfaces to Databases

[27] Leis et al., How Good Are Query Optimizers, Really?. PVLDB Volume 9, No. 3, 2015

[28] R. Zhao and K. Mao, Fuzzy Bag-of-Words Model for Document Representation. *IEEE Transactions on Fuzzy Systems*, vol. 26, no. 2, pp. 794-804, April 2018.

[29] US 9037464, Mikolov, Tomas; Chen, Kai & Corrado, Gregory S. et al., "Computing numeric representations of words in a high-dimensional space", published 2015-05-19, assigned to Google Inc.

[30] J. Pennington et al., GloVe: Global Vectors for Word Representation.

[31] Rajaraman, A.; Ullman, J.D. (2011). "Data Mining" (PDF). Mining of Massive Datasets. pp. 1–17

[32] Mannes, John. "Facebook's fastText library is now optimized for mobile". TechCrunch.

[33] D. Bahdanau, K. Cho, Y. Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate", 2014.

[34] Minh-Thang Luong, Hieu Pham, Christopher D. Manning, "Effective Approaches to Attention-based Neural Machine Translation", 2015.

[35] A. Lamb, A. Goyal, Y. Zhang, S. Zhang, A. Courville, and Y. Bengio. Professor Forcing: A New Algorithm for Training Recurrent Networks (2016), NeurIPS 2016.