Deep Learning for SQL Query Operators

Siksma, Casper S.

Award date:
2021

Department of Mathematics and Computer Science
Database Research Group

Deep Learning for SQL Query Operators

Casper S. Siksma

A thesis presented for the degree of
Master of Science

Supervisors:
dr. O. Papapetrou
dr. V. Menkovski

Committee:
dr. O. Papapetrou
dr. V. Menkovski
dr. N. Sidorova

version 3.2

October 19, 2021

Abstract

Despite recent advancements in database optimization, working with huge amounts of data remains a fundamental challenge. As a result, machine learning techniques are increasingly used for prediction problems and data-driven optimization when dealing with highly-dimensional data on a large scale. Where most approaches focus on improving some integral part of a database management system, such as query optimization or cost estimation, little research is known to exist on the prediction of future queries or their operators. In fact, the way that data is queried often corresponds to some user, application, or system behavior responsible for the queries. The goal of this work is to exploit this dependency to provide opportunities for optimization in a database management system.

This work proposes the concept of query execution plan fingerprinting, a novel encoding scheme for the feature engineering of SQL query data that aims to represent the intermediate results of a query in a lower dimension. Consequently, these features can be used by caching systems and predictive models to improve query performance. In addition, we present a deep neural architecture that enables the prediction of these features for subsequent future queries. This is done by formulating the prediction of encoded queries as a sequential sets to sequential sets learning problem.

For our experiments, the model architecture was implemented and trained on the historic query log of two distinct real-world databases. We test the effect of certain components and evaluate the model with a variety of methods and metrics. Our results reveal that good performance and generalization can be achieved for both databases. For one of the databases, the model outperforms the best performance of other methods with respect to recall by 46.8%-51.9%. However, in the other database, the model only outperforms other methods by 2.2-3.7% with respect to recall. Additionally, results show that increasing the number of elements that are predicted at each subsequent set negatively affects the precision of the model, highlighting a limitation of the model compared to other methods.

Further analysis examines the usefulness of the model in a practical setting. In particular, experiments were performed with a small fixed-size cache and a sequence of queries previously unseen by the model. Results demonstrate that Least-Recently-Used (LRU) and First-In-First-Out (FIFO) caching policies for the encoded query elements can be used to improve query answering time, resulting in speedups of up to 2.09x. In addition, the model can be used to improve upon these caching policies by keeping predictions cached, increasing query answering speed by up to 2.67x. Lastly, if the predictions of the model could be pre-computed before query arrival, speedups of up to 14.90x could be achieved.

Preface

First of all, I am grateful for the guidance I received from my supervisors, Odysseas Papapetrou and Vlado Menkovski. It was a difficult time for everyone due to the COVID-19 situation; staying at home for months on end and not being able to meet with friends and family made it difficult to stay motivated. Nevertheless, well-organized and frequent video meetings with my supervisors allowed me to discuss ideas, results and ask questions. I appreciate how easily I could approach either of them and receive help in a timely manner. After working together for over 6 months, it feels crazy to say that we have never met in person.

This thesis marks the end of my 7 year-long journey at TU/e. After completing my Bachelor's degree, I wasn't sure if I wanted to continue for another two years. Luckily, I was convinced to continue, and I don't regret it at all. The master's program in Data Science and Engineering has provided me with many great experiences, new knowledge, and a strong academic background for my future professional career.

Casper Sebastiaan Siksma
Eindhoven, The Netherlands
October 19, 2021

Contents

List of Figures

List of Tables

List of Abbreviations

DBMS	Database Management System
SQL	Structured Query Language
CRUD	Create, Read, Update, Delete
I/O	Input/Output
QEP	Query Execution Plan
LRU	Least-Recently-Used
FIFO	First-In-First-Out
SGD	Stochastic Gradient Descent
MLP	Multi-Layer Perceptron
RNN	Recurrent Neural Network
BPTT	Backpropagation Through Time
LSTM	Long Short-Term Memory
GRU	Gated Recurrent Unit
IoT	Internet of Things
ORM	Object-Relational Mapping
MSE	Mean Squared Error
BCE	Binary Cross-Entropy
NLL	Negative Log-Likelihood
WMSE	Weighted Mean Square Error
NDCG	Normalized Discounted Cumulative Gain
HR	Hit Rate
CPU	Central Processing Unit
GPU	Graphics Processing Unit
CNN	Convolutional Neural Network

Chapter 1

Introduction

As our ability to generate and capture data continues to explode, the size of the data landscape keeps growing. Not only does this data need to be stored, it also needs to be usable. However, making huge amounts of data usable gives rise to new challenges that require more efficient solutions. As a result, machine learning techniques are increasingly used for prediction problems and data-driven optimization when dealing with highly-dimensional data on a large scale. One such opportunity lies in query optimization in database systems. In particular, if it was possible to accurately predict future queries or components thereof, it would provide an opportunity to prepare for them. Consequently, query efficiency could be improved.

The optimization of database systems has been a prevalent topic in data and computer science research since their inception. However, the use of machine learning techniques in this domain has been a relatively recent topic of interest, with most research having been published in the past few years. Applications of machine learning in database systems range from query optimization [1]–[6] to query performance estimation [7] and learning query state representations [8]. Despite database optimization being a common theme, surprisingly little research exists on the prediction of queries or query operators [5]. This work will address the prediction of query operators with deep neural networks to provide a means of improving query efficiency in database systems.

The outline of this chapter is as follows. First, a short motivation is presented, followed by a summary of the research problem, the objectives of this thesis, and formulation of the research questions. Subsequently, the scientific contributions are highlighted and an outline for the rest of this work is given.

1.1 Motivation

The size of the worldwide data landscape is projected to be over 175 zettabytes[1] by 2025 and doubling in size approximately every two years thereafter [9]. As a result, databases are growing too large to be trivially handled with today's computers, this is commonly referred to as *data deluge*. To store and work with these massive amounts of data, distributed database systems have become an increasingly necessary solution [10]. However, it remains a challenge to keep up with the ever-increasing volume of data. As opposed to further vertical or horizontal scaling by upgrading hardware and using more servers, one could also try to optimize the ways that these systems store and handle data.

One such opportunity lies with the querying of database systems. In particular, the way that data is queried often corresponds to some user, application, or system behavior responsible for the queries. It is this dependency that could potentially be exploited to provide opportunities for optimization. Database query logs often exhibit these patterns, where one query gives some indication of the next query [5], [11]. For example, a common pattern might be querying a product's ID using the product's

[1]A zettabyte is 10^{21} bytes

name, followed by querying the number of products in inventory using that product's ID. A query for the latest social media posts may also want to request information of the authors, so reading of a 'posts' and a 'users' tables could frequently co-occur in the same query. An inventory management system for a restaurant chain may also frequently query the stock of products that are related to each other, such as hamburgers and buns for a particular venue.

If future queries could be predicted, a database management system (DBMS) could prepare for them in advance. For instance, a DBMS could preemptively build an index, compute a synopsis, pre-execute a join, or keep something cached in memory. Then, when a new query arrives that was expected, the result is already (partially) in memory and the query can quickly be answered. Most, if not all, of the computation for the query could already be finished before the query arrives. Furthermore, accessing the file system, which is generally orders of a magnitude slower than accessing memory [12], could be done preemptively or made entirely redundant by caching (intermediate) results from a previous query. The sequence diagram for the example of a restaurant chain is shown in Figure 1.1. If it was known that the second query was very likely to follow the first, it could make sense to keep the `product_stock` table, filtered on `city='eindhoven'`, in memory, as opposed to reading it from disk again.

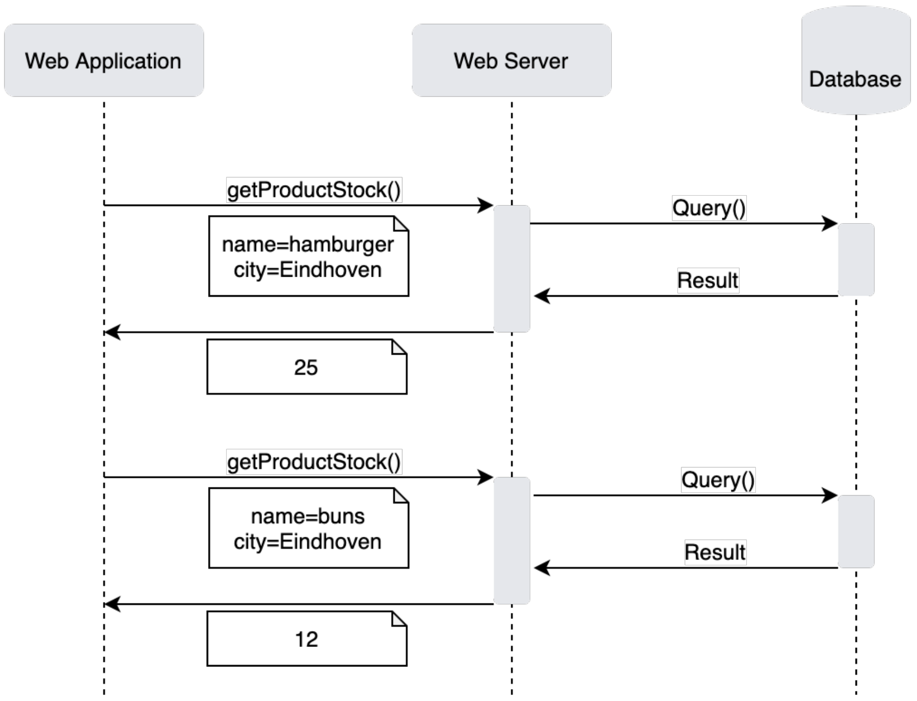


Figure 1.1: A sequence diagram for an example query pattern

Consider also the case of large distributed systems, where latency cost between data centers and edge-nodes is a significant factor. Edge-nodes are servers that are geographically close to the source of the query and may be a long distance away from the data centers that store the required data. Take for example the distributed system maintained by Google, see Figures 1.2 and 1.3. Not only are there many more edge nodes than data centers, but edge nodes could be a long distance away from data centers [13]. Preparing for incoming queries on edge nodes would thus eliminate the latency cost of sending the query and the result over the network for a particular query. A predictive model for queries would enable this form of edge computing, where computation and data storage are brought closer to the source of the query to improve response times and save bandwidth.

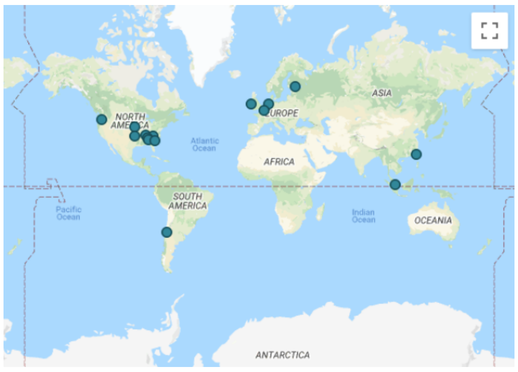


Figure 1.2: Google's distributed database system data center locations [13]

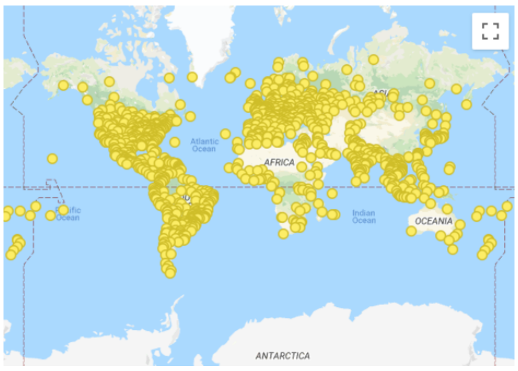


Figure 1.3: Google's distributed database system edge node locations [13]

1.2 Research Problem

The rest of this study will investigate deep learning as a means of enabling database optimizations by creating a predictive model for future queries based on historic query data. A distinction is made between two sub-problems, the feature engineering of Structured Query Language (SQL) query data and the conception of a predictive model.

1.2.1 SQL Query Encoding

The first problem revolves around the feature engineering of SQL queries. Queries need to be encoded such that predictions can be *meaningful*. The queries in the log of a database system are sequential, and recurrent neural networks are especially applicable to sequential data to create predictive models [14]–[17]. However, the use of neural networks requires the input to be vectorized, so an encoding of SQL queries is needed. Ideally, we would be able to predict the query in its entirety. But, the high dimensionality of SQL queries would result in too many parameters to train a model efficiently. Nevertheless, queries also include information on the tables, attributes, and parameters involved. Hence, being able to accurately predict these would already enable preparing for future queries in a variety of ways. Therefore, the first challenge is determining how to encode (vectorize) SQL queries, such that machine learning techniques can be used, and a (partially) correct prediction can be used for query preparation. In addition, given a query log and the database schema, we should be able to encode this query log into a sequence of encoded queries.

Objectives

Consequently, we identify the following objectives in addressing this first research problem.

- Design an *encoding scheme* for encoding queries. The goal is to encode each query into a low-dimensional representation that captures as much information as possible in order to make meaningful predictions. A prediction is considered meaningful if it could be used to improve query efficiency and/or performance by reducing computational costs, latency, bandwidth usage or I/O operations.
- *Pre-process* the query log by extracting the raw SQL queries, altering the queries based on SQL dialect such that they can be run by our processor, and filtering non-SELECT queries from the log. Missing information and placeholders should be replaced and problematic or corrupt queries should be removed.
- Develop an *encoding framework* that implements the designed encoding scheme. The encoding framework should be able to load the schema and corresponding query log of a database, and

subsequently encode every query. The framework should be able to handle different databases with different schemas and SQL dialects. In addition, the encoding framework will be used for analytics and it should support exporting the encoded queries to disk as a usable data set, which in turn can be used as input for machine learning frameworks such as PyTorch.

Research Question 1

Based on the above we present the first research question:

How could SQL queries be encoded into lower dimensional features such that feature predictions of future SQL queries would enable optimizations in database management systems?

1.2.2 Prediction Model

The second problem is predicting the query operators of future queries based on the encoded historic data. Given a sequence of encoded queries, we would like to predict the next sequence of encoded queries or components thereof. It follows that we need to induce a sequence to sequence model based on the encoded query log. To formalize this problem, given a sequence of encoded queries $\{x_1, x_2, \ldots, x_l\}$, we would like to have a model of the joint distribution of the next sequence $\{y_1, y_2, \ldots, y_m\}$, as is shown in Equation 1.1. Consequently, given an input sequence of encoded queries $\{x_1, x_2, \ldots, x_l\}$, we could use the model to find the most likely sequence of encoded queries $\{y_1, y_2, \ldots, y_m\}$ to succeed it, see Equation 1.2.

$$\mathbb{P}[\{y_1, y_2, \ldots, y_m\} \mid \{x_1, x_2, \ldots, x_l\}] \tag{1.1}$$

$$\underset{y_1, y_2, \ldots, y_m}{\arg\max}\ \mathbb{P}[\{y_1, y_2, \ldots, y_m\} \mid \{x_1, x_2, \ldots, x_l\}] \tag{1.2}$$

Objectives

To address the second research problem, the following objectives were identified.

- Design and implement a *model architecture* to model the joint conditional probability of the output sequence. Parameters such as input sequence length, output sequence length and dictionary size should be adjustable by the user in order to support different databases and optimization goals.
- *Training and optimization* of the model using the historic query data to obtain a performant model. Appropriate training, validation and test sets need to be created from the encoded query log to measure the model's capability to generalize to data outside the training set. Additionally:
  - *A* loss function needs to be determined to quantify predictions.
  - *Class imbalance* needs to be addressed to avoid low predictive accuracy of infrequent classes.
  - *Hyperparameter tuning* of optimizer algorithms, learning rate, embedding size, number of epochs etc. to improve model performance
  - *Regularization* should be used in an attempt to improve model performance and avoid overfitting.
- Implementation of an *inference* algorithm to obtain the most likely sequence of encoded queries given a sequence of encoded queries. Inference should not be too computationally expensive to provide predictions in a timely manner.
- *Evaluation* of the model using suitable metrics and comparison with other prediction methods.

- Investigate the model's *usefulness* by placing the model in a practical setting and measuring its effect on query answering time, i.e. the time needed to answer a query.

Research Question 2

This leads us to the second research question:

How could a deep learning based framework be used to build a predictive model for SQL query data?

1.3 Contributions

The scientific contributions of this thesis are two-fold.

First, the current state of the literature is limited regarding the prediction of database queries or components thereof, and typically use SQL (text) based approaches. This work presents a novel approach for the feature engineering of SQL queries for machine learning. Namely, by generating the corresponding query execution plan and splitting it into a set of reusable components, the query operators, we enable predictions that are partially correct. Thereby, this work presents a method of encoding these query operators into vectors of dimensions suitable for neural networks and possibly other predictive methods.

Second, while machine learning has been used to create query optimizers or estimate query execution time, little research exists on machine learning as a means of creating a *predictive* model for database queries. This thesis demonstrates a predictive model based on a deep learning framework for query operators, by which the operators of future queries can be predicted given the operators corresponding to queries in the past. In turn, the predictions could be used by a database management system to prepare for future queries and improve overall query efficiency.

1.4 Outline

The remainder of this work is organized as follows. For the ease of the reader, background information and related work on database management systems and machine learning is discussed first in Chapter 2. Subsequently, Chapter 3 presents the proposed method for encoding SQL queries into vectors. Afterward, Chapter 4 describes the conception of the machine learning model with a deep learning framework. This includes the architecture of the model and how it can be used for inference. Chapter 5 presents the experimental setup and includes information on what data sets were used, how the data was prepared, and how the model was trained and evaluated. Chapter 6 presents and discusses the obtained results. Finally, the work is concluded in Chapter 7 together with the known limitations and suggestions for future work.

Chapter 2

Preliminaries

This chapter introduces background information and related work on database management systems and machine learning to aid in understanding the contributions of this work. First, the reader is presented with a brief introduction to database management systems with a special focus on query execution plans. Subsequently, we present the reader with existing literature relevant to the feature engineering of SQL queries. Afterward, we introduce the field of machine learning and how it could be applied to our problem; what constitutes a learning problem, and how we formalize the process of learning. Then, we briefly describe (deep) neural networks as a class of prediction rules and focus on a specific type of neural networks for sequential data.

2.1 Database Management Systems (DBMS)

A database management system (DBMS) is software for creating and maintaining databases and enables end-users to perform create, read, update, and delete (CRUD) actions on data in the database. DBMSs generally offer support for one particular database model, such as relational (e.g. PostgreSQL), document (e.g. MongoDB), and key-value (e.g. Redis) databases. From the existing database models, relational databases are by far the most common [18]. Recently, big data platforms such as Apache Spark also added support for working with relational data [19], [20]. Relational databases were first introduced in the 1970s [21] and later popularized in the 1980s by providing a convenient way of modeling data as rows and columns in a set of tables. The *relational* aspect comes from the way that these tables are interconnected. For example, a restaurant chain may have multiple venues, and customer orders are associated with one of the venues at which they were placed. A relational database models this relation by associating rows in one table with rows of another. An example is given in Tables 2.1 and 2.2, where orders have an associated `venue_id`.

id	items	price	venue_id
1	{cola: 2, beer: 7}	21.0	1
2	{coffee: 1}	2.5	4
3	{fries: 2, beer: 2}	14.0	2
4	{nachos: 1, wine: 6}	29.0	2

Table 2.1: **Order**

id	name	city
1	Eindhoven	Eindhoven
2	Eindhoven Strijp	Eindhoven
3	Eindhoven Station	Eindhoven
4	Veldhoven Noord	Veldhoven

Table 2.2: **Venue**

To perform CRUD actions, queries are generally formulated in the Structured Query Language (SQL), the de facto standard of query languages in relational databases. These queries are evaluated by the query processor to obtain the desired result. For example, one could retrieve all the venues located in Eindhoven using `SELECT * FROM venue WHERE city='Eindhoven'` or retrieve the ordered items from orders that have a total price over 25 euros with `SELECT * order WHERE price>25`. Obtaining the results of these queries is relatively simple by scanning over each row in the table and comparing the query parameter with the row attribute values. However, SQL allows for more complex operations, such as joining multiple tables together. Consider the following query in Algorithm 1, whose goal is to obtain the order IDs from all the orders placed at the venue named 'Eindhoven' with a price greater than or equal to 20 euros.

Algorithm 1: Example SQL Query

```
1: SELECT order.id
2: FROM order, venue
3: WHERE order.venue_id = venue.id
4: AND venue.name = 'Eindhoven'
5: AND order.price ≥ 20
```

In this case, the order of operations is not immediately clear. One option is to find the venue ID of the venue named 'Eindhoven' first, and then all orders associated with that venue ID and a price greater or equal to 20 euros. Another option is to filter all the orders whose price is less than 20 euros first, and then for each order check if the associated venue ID corresponds to a venue named 'Eindhoven'. This problem becomes worse the more tables are involved in the query. To illustrate, determining the join order of a join over 10 tables already results in 3,628,800 possible join orders, as there are $n!$ possible combinations of join orders for a join over n tables [22]. Naturally, some orders of operations are much more efficient than others. Therefore, most DBMSs implement a query processing pipeline that aims to optimize this process by finding the most efficient order of steps to take to answer the query. Such a query processing pipeline is outlined in Figure 2.1.

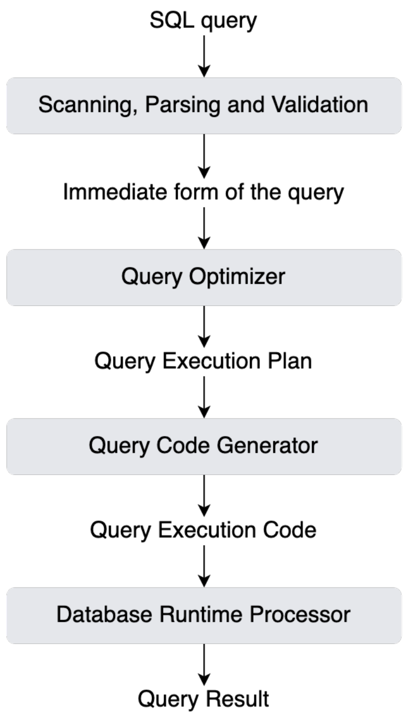


Figure 2.1: Query Processing Pipeline

In particular, the *query optimizer* is responsible for finding the optimal sequence of evaluating the operators and accessing tables for a given SQL query. The result is what is known as the query execution plan (QEP).

2.1.1 Query Execution Plans (QEP)

For a given SQL query, the query execution plan (QEP) can be defined as the sequence of steps performed by the DBMS that are required to evaluate the query. For the example query in Algorithm 1, Figure 2.2 shows a corresponding QEP. A DBMS might generate multiple QEPs for a query and execute only the one with the lowest cost estimate. This particular QEP was determined to be the most optimal by PostgreSQL, a popular open-source relational database management system, when executing the query on the database. As can be seen in Figure 2.2, the optimal (as estimated by the query optimizer) way of evaluating this query is by first reading the order and venues table separately, and filtering them based on order price and venue name respectively. Subsequently, the venue IDs are matched between orders and venues. Lastly, the result is projected on the order ID attribute to only obtain a list of order IDs.

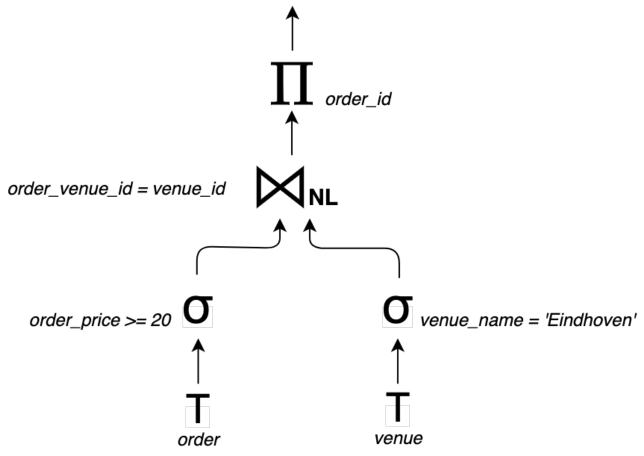


Figure 2.2: A QEP for the example SQL query in Algorithm 1

For a given SQL query q let us now formally define the QEP. Let p be the QEP of q, which is a tree where each non-leaf node is an operator from a set of operators $\mathcal{A}$. For example, a nested loop-join is denoted by $\bowtie_{\text{NL}}$ and a union is denoted by $\cup$. In addition, let each leaf node of p be a base table access operator, either a table scan or an index scan which we denote by **T** and **I** respectively. Thereby, the example QEP in Figure 2.2 could be represented as follows, where parentheses indicate the operator hierarchy:

$$\left(\sigma_{\text{order\_price}\geq 20}\big(\mathbf{T}(orders)\big) \bowtie_{\text{NL}} \sigma_{\text{venue\_name}=\text{`Eindhoven'}}\big(\mathbf{T}(venues)\big)\right)$$

Throughout this thesis, we will use the commonly used relational algebra symbols to denote certain operators, such as 'Π' for projections. However, there is not a one-to-one relation between QEPs and relational algebra, as the QEP may include operators that are undefined in relational algebra, such as operators for `ORDER BY` and `LIMIT` clauses [23]. This is also the reason we define additional operators **T** and **I**, as no distinction is made between table and index scans in relational algebra.

2.2 The Feature Engineering of SQL Query Data

Unfortunately, research on the feature engineering of SQL queries for the purpose of creating a predictive model is limited. Perhaps the only research similar to the problem domain is research introducing Apollo, a learning system of query correlations for predictive caching in geo-distributed systems [5]. They propose the use of query templates aggregated into a frequency-based Markov graph to define dependency relationships between queries. Query templates are constructed from a query by replacing the parameters with a placeholder such that if two queries are identical except for their parameters, they correspond to the same query template. For example, the query '`SELECT`

`* FROM students WHERE id=197283`' becomes '`SELECT * FROM students WHERE id=?`'. While this addresses the complexity of query parameters by focusing only on the structure of the query, the problem with this approach is that a small change in the query still results in a completely different template. For example, by adding an `ORDER BY` clause to the query, its template changes despite only the ordering of the results being different. This also means that a predictive model needs to predict a query in its entirety to have a prediction that is both valid and useful, i.e. there are no partially correct predictions. Thus, using the SQL query templates as features greatly limits the potential of any predictive model.

Therefore, we can instead look towards using the query execution plan (QEP) of a query as input of our encoding scheme. The QEP includes all the query details while providing a hierarchical structure. Several methods to encode QEPs have been demonstrated in previous research, albeit for different purposes. These purposes include, among others, query optimization and query cost estimation [4], [6], [7]. For example, Neo and Bao are query optimizers that leverage reinforcement learning [4], [6]. Regardless of the application, using QEPs in neural networks requires vectorization, so we could draw inspiration from these publications.

One approach of encoding QEPs is to use two different encodings: a query encoding for the query details independent of the QEP, and a plan encoding for the operations of the QEP [4]. In the model, these two encodings can be concatenated to obtain their joined representation. An example of such an encoding can be seen in Figure 2.3, where an adjacency matrix for the joins as well as a vector indicating the presence of a column are combined into a single query level encoding. While this

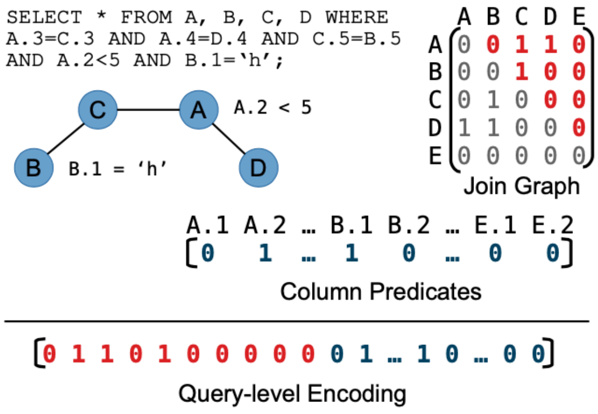


Figure 2.3: Query level encoding [4]

query level encoding conveys information on what tables are joined and what column predicates are involved, it fails to convey information on the query operators. That is, this encoding does not include information on the operators except for what tables are joined. It fails to capture information on the type of join, the order of operations, and neglects other operators, such as projections and filters.

Another option is to encode each node of the QEP tree into a vector [4], [6]. Constructing these vectors is done based on the existence of a certain operator, table, or index and yields a tree of vectors. Figure 2.4 demonstrates such an encoding. One benefit of keeping the tree structure intact is that tree operations (e.g. tree convolutions) could be used in the model [4], [6]. Alternatively, the tree could be flattened into a single vector as was also demonstrated in other works [1], [24]. However, while this encoding includes information on the operators, it loses the information on the table attributes, such as the attributes used in joins, filters, and projections. Using a machine learning-based predictive model gives rise to another problem with this approach. In particular, this type of encoding scheme is susceptible to 'grammar' errors as the QEPs are predicted based on probability. For example, an *incorrect* prediction might indicate joins on tables that can not be joined, index scans of tables for which no index exists, and leaf nodes not corresponding to table/index scans etc. Including table attributes in this encoding scheme would make predictions even more susceptible to errors. Previous research also assumed that QEP trees were strictly binary, while this need not always be the case [6]. For instance, operators for sorting only have one child node, and union operators can have more than two child nodes.

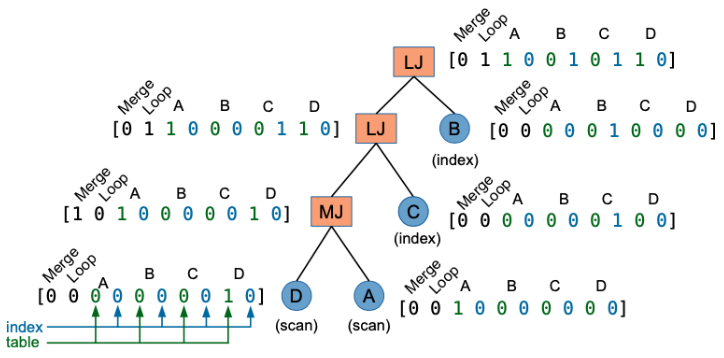


Figure 2.4: Plan level encoding [4]

One could also look at research in other domains where the encoding of graph data played a key role. A notable example is the application of neural networks on molecular structures [25]. In their paper, they introduce an algorithm that uses a hashing approach to encode an atom and its neighbors into an integer index. This allows them to encode the highly dimensional structure of molecules into a relatively small integer representation that is suitable for use in neural networks and end-to-end prediction pipelines.

2.3 Machine Learning

Machine learning is part of the broader field of artificial intelligence. Specifically, machine learning focuses on the research and development of algorithms that allow computers to perform certain tasks by learning from data without being programmed how to do those tasks explicitly [26]. One particular class of algorithms is that of neural networks, which vaguely mimic human brain activity by connecting artificial neurons to recognize relationships between large amounts of data. *Deep learning* is a subset of *machine learning*, where the 'deep' refers to the use of multiple layers in these networks resulting in deeper model architectures. Ultimately, deep learning is one of many techniques that can be used to address the problem of *learning*.

However, what constitutes a learning problem? Many of the systems that are developed today are intended to optimize processes and enable automation. Building these solutions requires defining a *set of rules* such that the guaranteed outcome satisfies some kind of goal. In addition, these solutions can become arbitrarily complex as the set of rules grows and the goals become more ambitious. Consider the process of building a skyscraper as an example. Building a skyscraper requires careful planning, calculations, heavy machinery and is labor-intensive. The set of rules that would guarantee that the outcome of the process is the skyscraper as per the design is tremendous. Regardless of how complex the process of building a skyscraper may seem, this does not constitute a learning problem as there is sufficient understanding to meet the desired goal. In other words, any algorithm that learns from observation would not outperform the existing solution.

On the other hand, many problems that are easy for humans are hard to solve algorithmically. Imagine the scenario of detecting cats or dogs in an image, for which some examples are shown in Figure 2.5. This is trivial for any human, yet very complex for any algorithm when you consider the number of ways in which combinations of pixel values determine whether a picture contains a cat or a dog. In this case, the set of rules is too large to fully express the solution to our problem. However,

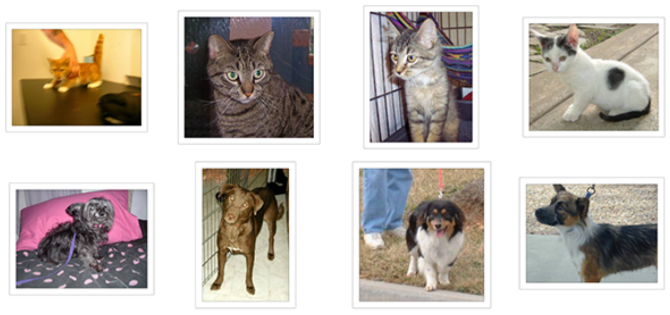


Figure 2.5: Select images of cats and dogs [27]

we might be able to use the distribution of pixel values to determine whether the image is more likely to portray a cat or dog, as will be further elaborated on in the next section. Comparing this example to the previous example of building a skyscraper, the distinction can be made between writing a set of rules and learning a set of rules.

2.3.1 The Learning Problem

To formalize a learning problem we use concepts from statistical learning theory [28], [29]. Let us again consider the problem of classifying pictures of cats and dogs. The goal would be to construct a model that can predict whether the images contain either a cat or a dog. Specifically, assume the images have a resolution of 256×256 pixels and when one of these images is given as input to our model, it should return either 'cat' or 'dog'. In supervised machine learning, models are constructed automatically by feeding it a subset of the images and the accompanying ground-truth label, the training data set. Equally important, the model should generalize well to pictures outside the training data set, i.e. images not seen before by the model, which is evaluated using a validation and testing set. Thus, the model should be able to learn from the data that it receives and make decisions based on the learned information. It should be able to apply what it has learned to data it has never seen before. In unsupervised machine learning, the labels are incomplete or missing in their entirety, and finding structure in the data according to the features is also left up to the model, this is also known as clustering.

For any prediction task, it is necessary to learn from the data space, which is defined as

$$\mathcal{X} = \text{Feature Space}$$

$$\mathcal{Y} = \text{Label Space}$$

The feature space specifies all possible values of the data, while the label space specifies all possible labels of the instances in the feature space. The label space accompanies the feature space such that for each $x \in \mathcal{X}$ there is a $y \in \mathcal{Y}$ denoting the label of x. Consider the example of classifying pictures of cats and dogs. Assuming the images are grayscale and each pixel has an intensity between 0 and 255, the feature space would then be $\mathcal{X} = [0, 255]^{256\times256}$, and the label space would be $\mathcal{Y} = \{\text{'cat'}, \text{'dog'}\}$. For a prediction task, we typically observe an instance of the feature space and try to predict its label in the label space. More formally, we want to determine a mapping $f : \mathcal{X} \to \mathcal{Y}$ that predicts $y \in \mathcal{Y}$ for any given input $x \in \mathcal{X}$ as well as possible. However, defining what is a good prediction and what is not depends on the problem and the available data.

Empirical Risk Minimization

To evaluate a prediction we use what is called a loss function. Suppose there is an instance $x \in \mathcal{X}$ with an accompanying label $y \in \mathcal{Y}$. Now let $\hat{y} \in \mathcal{Y}$ be the predicted label after processing x. The loss function is then defined as a mapping $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$. Generally, the loss function that is used depends on the relevant learning problem. A simple example of a loss function is the 0/1-loss, given in Equation 2.1, where the only distinction being made is between a correct and incorrect prediction.

$$\ell(\hat{y}, y) = \begin{cases} 1, & \text{if } \hat{y} \neq y \\ 0, & \text{otherwise} \end{cases} \tag{2.1}$$

Using the statistical learning framework, we assume that the combinations of features and labels are samples from a joint probability distribution on $\mathcal{X} \times \mathcal{Y}$, denoted by $\mathcal{D}$. Accordingly, let $(X, Y) \in \mathcal{X} \times \mathcal{Y}$ denote a pair of random variables distributed according to $\mathcal{D}$. Even though every combination of feature and label (X, Y) is possible in theory, there are some combinations that are more likely than others. For instance, the pictures of cats and dogs will almost never look like a uniform gray color. Hence, some combinations of pixel values are more likely than others. The objective is then to learn a function $f : \mathcal{X} \to \mathcal{Y}$ that minimizes the expected loss, also known as the statistical risk:

$$R(f) = \mathbb{E}\big[\ell(f(X), Y))\big] \tag{2.2}$$

The statistical risk $R(f) \to \mathbb{R}$ indicates how a mapping $f : \mathcal{X} \to \mathcal{Y}$ performs with respect to the loss function ℓ. However, choosing f such that the risk is minimized is not trivial. To compute the risk, we need to know the joint distribution $\mathcal{D}$, but $\mathcal{D}$ is unknown. Therefore, it is possible to use the available data based on the assumption that they contain representative samples from $\mathcal{D}$. In particular, let $D = \{X_i, Y_i\}_{i=1}^n$ denote the available data. Consequently, the risk can be minimized over the data D to find a mapping f whose risk approximates the risk over $\mathcal{D}$. This is defined as empirical risk minimization, where the empirical risk is given by:

$$\hat{R}_n(f) = \frac{1}{n} \sum_{i=1}^{n} \ell(f(X_i), Y_i) \tag{2.3}$$

In other words, the empirical risk is the average loss for all the samples in our data set for a rule f. It follows that we would like to choose a prediction rule based on minimizing the empirical risk. Hence, given a class of functions $\mathcal{F}$ we can define the empirical risk minimizer

$$\hat{f}_n = \arg\min_{f \in \mathcal{F}} \hat{R}_n(f).$$

Even though this is the best we can hope to do without making assumptions about the joint distribution $\mathcal{D}$, there is a danger to this approach. Consider the case where the collection of candidate prediction rules $\mathcal{F}$ is extremely large, possibly infinite. A large $\mathcal{F}$ would mean that we can always make

$$\inf_{f \in \mathcal{F}} \hat{R}_n(f)$$

small, given that we are increasing the number of ways to fit the data. Thereby, one could choose a rule $\hat{f}_n \in \mathcal{F}$ such that $\hat{R}_n(f)$ is small, or 0 even if the rule perfectly fits the data. Unfortunately, this does not mean that this prediction rule performs well for instances *outside* of the data set. This problem fits the description of *overfitting*. However, the real problem is the model not being able to generalize outside the data set, as there could exist rules that generalize well despite overfitting the training data[1]. In an attempt to address this problem, we split the data set D into three disjoint data sets: a training set D_{train}, a validation set D_{val}, and a test set D_{test}. D_{val} is used to evaluate f during training, and D_{test} is used to evaluate f after training completes. This way, we can estimate how well f generalizes to data outside D, i.e. previously unseen data.

[1]1-Nearest Neighbor or some deep neural networks are sometimes able to fit the data perfectly while still generalizing well outside the training data [28]

2.3.2 Neural Networks

The work in this thesis focuses on the use of deep learning as a means of addressing a learning problem. Therefore, the class of models $\mathcal{F}$ that will be considered is that of neural networks. The concept of neural networks will be briefly described for those unfamiliar with this class of prediction rules. First, we will focus on the elementary units of neural networks, the artificial neuron. Subsequently, we will demonstrate how these neurons can be used to learn from data and how they can be connected to create networks and learn more complex representations. Lastly, a specific class of neural networks that are particularly suitable for our problem will be discussed.

The Artificial Neuron

Inspired by biological neurons found in the brain, the artificial neuron receives one or more (weighted) inputs and produces one output by means of some 'activation' function to the weighted sum. A visual representation of an example artificial neuron can be seen in Figure 2.6.

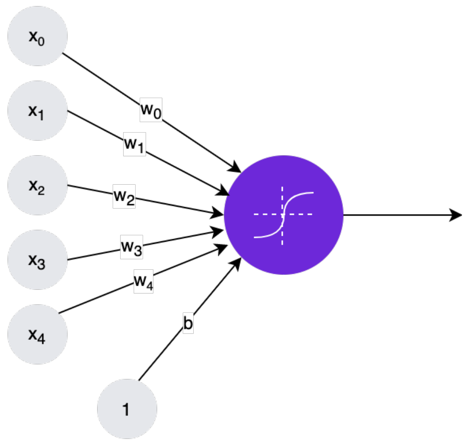


Figure 2.6: Artificial Neuron

More formally, this particular neuron takes 5 inputs which are given by x_0, x_1, x_2, x_3, x_4, also denoted as input vector $\mathbf{x}$. The edges between the nodes are parametrized by the weights vector $\mathbf{w}$ and a bias term b. The weights and bias together form the parameters of the neuron, denoted as θ. When the neuron 'activates', some function ϕ is applied to the weighted sum of the inputs offset by the bias. For any artificial neuron we can denote its output[1] $o_\theta(x)$ by

$$o_\theta(x) = \phi\Big(\sum_i w_i x_i + b\Big) = \phi\big(\mathbf{w}^\top \mathbf{x} + b\big).$$

Observe that the artificial neuron becomes a linear regression model when we use a linear activation function:

$$o_\theta(x) = \mathbf{w}^\top \mathbf{x} + b$$

Recall how we cast the learning problem in the framework of statistical learning theory, and let our class of models $\mathcal{F}$ be defined as the set of neural networks with only one neuron that takes exactly 5 inputs. In this case, we would like to choose parameters θ such that the empirical risk is minimized. Let $\hat{\theta}$ denote the empirical risk minimizer, then we have

$$\hat{\theta} = \underset{\theta}{\arg\min}\, \hat{R}_n(o_\theta) = \underset{\theta}{\arg\min}\, \frac{1}{n}\sum_{i=1}^{n} \ell(o_\theta(\mathbf{x}_i), y_i). \tag{2.4}$$

Thus, to find a neuron that best fits the data, we choose the parameters θ such that the empirical risk is minimized. However, the problem of choosing these parameters still remains.

[1]In matrix notation $\mathbf{w}^\top \mathbf{x}$ denotes the dot product of vectors $\mathbf{w}$ and $\mathbf{x}$

Gradient Descent

The minimization in Equation 2.4 needs to be implemented explicitly using an algorithm. A commonly used optimization algorithm is *gradient descent*, which allows us to optimize parameters for models that far surpass the complexity of linear regression models. The concept of gradient descent is not difficult to understand. Consider any continuously differentiable function $f : \mathbb{R}^2 \to \mathbb{R}$, which when plotted looks like a hilly landscape. If we start at any point on this landscape, the logical step to take is in the direction with the biggest decline, i.e. in the direction of the gradient of f. This process is repeated until a local minimum is reached and the loss function does not decrease any further. A visual representation of this process is given in Figure 2.7.

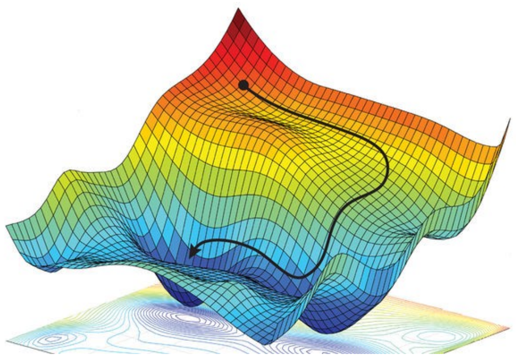


Figure 2.7: Gradient descent optimization for finding a local optimum [30]

In machine learning applications, we consider the empirical risk as a function of the parameters and step in the direction where the empirical risk declines the most. The gradient of the loss is computed with respect to all parameters in the model (weights and bias) and the parameters are updated to 'take a step'. In practice, it is common to use a stochastic approximation of gradient descent, called stochastic gradient descent (SGD), which estimates the gradient rather than computing it to reduce the computational burden.

Classification

In many cases, we would like to assign a discrete output to our inputs. To predict a discrete value from a fixed set of classes, rather than predicting a continuous value, we can use an activation function whose output has the same properties[1] as a probability distribution over a discrete set of elements. Two notable examples are the *logistic sigmoid* function for binary classification and *softmax* function for multiclass classification. For the example neuron in Figure 2.6 and the sigmoid function we can compute the output as follows:

$$\begin{aligned} o_\theta(x) &= \text{sigmoid}(w_0x_0 + w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b) \\ &= \text{sigmoid}(\mathbf{w}^\top \mathbf{x} + b) \\ &= \frac{1}{1 + e^{-(\mathbf{w}^\top \mathbf{x} + b)}}. \end{aligned}$$

For binary classification, such as in our example of classifying images containing a cat or dog, we can then simply define the probability of the input belonging to either output label as

$$\mathbb{P}(y = \text{'cat'} \mid \mathbf{x}) = o_{\text{cat}} = o_\theta(\mathbf{x}) = \text{sigmoid}(\mathbf{w}^\top \mathbf{x} + b)$$

and

$$\mathbb{P}(y = \text{'dog'} \mid \mathbf{x}) = o_{\text{dog}} = 1 - o_{\text{cat}}.$$

[1]Takes values between 0 and 1, its components are positive, and in the case of the sigmoid function also sum to 1.

Multi-Layer Perceptron (MLP)

So far, we have shown how a single neuron can learn from data by updating the weights of the inputs with respect to the loss function. Additionally, we have shown how the output of a neuron can be used to classify instances of the data. However, there is a limitation to the artificial neuron. Namely, a single neuron develops a linear combination of the weights which describes a decision (hyperplane) boundary that partitions the input space into two regions corresponding to the two labels. This limitation prevents us from modeling more complex and non-linear mappings between the input and output. To address this, we can combine neurons by using the output of one neuron as the input of another. Furthermore, by stacking neurons we can form *hidden layers* between the input and output layers. The result is the so-called Multi-Layer Perceptron (MLP), an artificial neural network, of which an example is given in Figure 2.8 below.

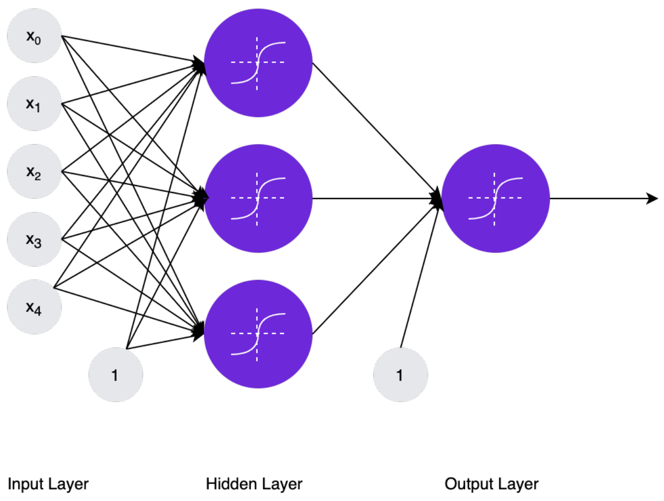


Figure 2.8: Example of an MLP with a single hidden layer

Observe that each neuron receives the same input values and the output of the hidden layer becomes the input of the next layer (in this case the output layer consisting of a single neuron). The neurons in the same layer typically have the same non-linear activation functions. Perhaps not surprisingly, we can also stack multiple layers to create *deeper* networks, which enables us to decompose decision boundaries into a set of simpler decision boundaries. An example of an MLP with two hidden layers is given in Figure 2.9.

To fit multi-layer neural networks with gradient methods we also need to compute the gradient of the loss. Rather than computing the gradient with respect to each weight individually, a *backpropagation* algorithm computes this efficiently by calculating the gradient per layer going backward from the last layer to the first [31], demonstrated in Figure 2.10. The computed gradients of one layer are reused for the computation of the previous layer to improve learning efficiency.

2.3.3 Sequential Models

Data can also be subject to temporal correlations. Temporal correlations in natural language, for example, show that some words tend to precede other words. The word suggestions on modern smartphone keyboards are a demonstration of this fact. Some other examples of domains with sequential data are electrocardiograms, DNA sequencing, speech recognition, and financial markets.

The main challenge is being able to detect (long) signals in the data. Any kind of model should ideally not need to see the whole signal to produce the output. Using a sliding window approach, where the contents inside the window form the input of the neural network, one could use a window size roughly equal to the size of the signal to perform this task. However, this quickly becomes

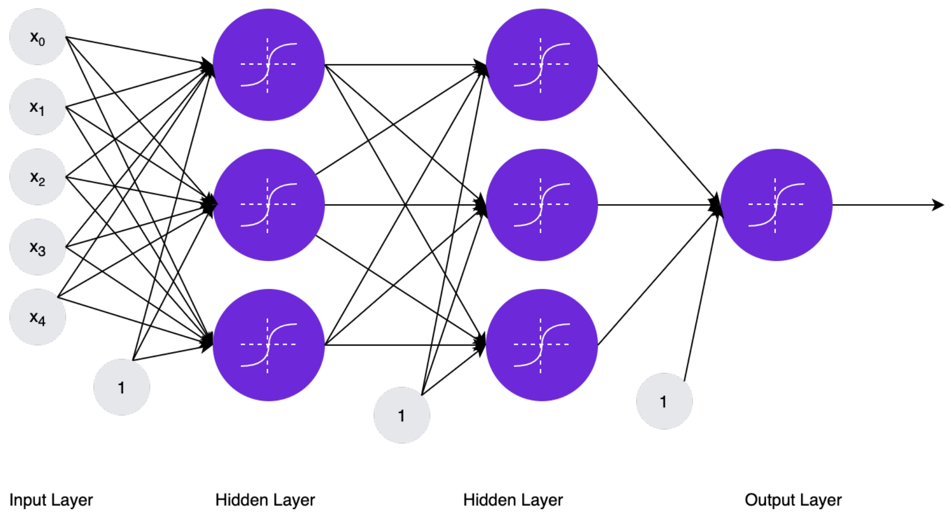


Figure 2.9: Example of an MLP with two hidden layers

impractical due to the vast number of ways data can be arranged inside the window, as signals can be very long. Alternatively, one could try using a small sliding window that can detect key points of the signal, such as the start and end. Unfortunately, this is not possible due to the fact that our neural network now needs *memory*. In particular, the start of the signal, when detected, should be memorized until the end of the signal is detected to identify the signal in its entirety. The neural networks as they have been presented thus far are incapable of storing information between consecutive inputs. Consequently, the *recurrent neural network* (RNN) was introduced in the 1980s [32].

Recurrent Neural Networks (RNN)

Recurrent neural networks (RNN) are a type of network that can also learn to model the temporal relation between inputs [32]. In other words, RNNs have a temporal dimension. Additionally, RNNs can process variable-length sequences of data points at one time while producing an output at each step. The temporal dimension is captured in the form of a memory state at each step. Thereby, RNNs are well suited for data that exhibit temporal correlations and can be used for (aligned) sequence classification, sequence generation, sequence to sequence modeling, and more.

The RNN architecture is similar to that of the MLP but introduces an internal *hidden state* with links and a delay, illustrated in Figure 2.11. We can unfold the RNN in time to show the distinct time-steps t, demonstrated in Figure 2.12. Weight matrices W, U and V maintain input-to-hidden, hidden-to-hidden and hidden-to-output weights respectively. The hidden states are high-dimensional vectors and create a recurrence by being connected to each other. Thereby, the output of the hidden state h_t of one time-step is used as input of the next time step, such that

$$h_t = f_h(x_t, h_{t-1}).$$

Furthermore, we have that the output y_t of one time-step is defined as a function of the hidden state:

$$y_t = f_y(h_t).$$

More formally, we can define the RNN cell as shown in Figure 2.13. Let weight matrices W, U, V and biases b, c be parameters of the model, and ϕ_1, ϕ_2 to be activation functions. With ϕ_1 = tanh

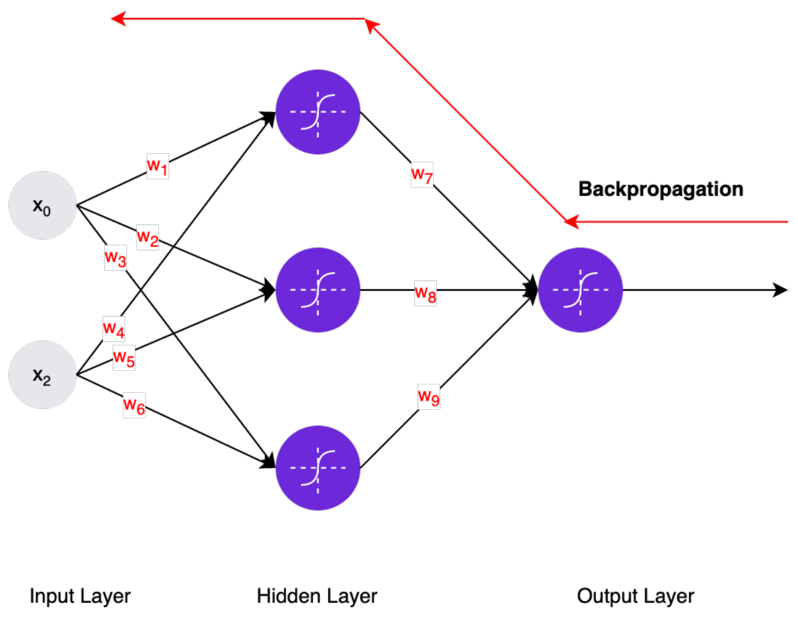


Figure 2.10: Backpropagation in an MLP with two input neurons, three hidden neurons, and one output neuron (biases are omitted for brevity)

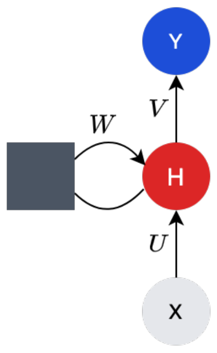


Figure 2.11: RNN architecture, where the square connection indicates a delay.

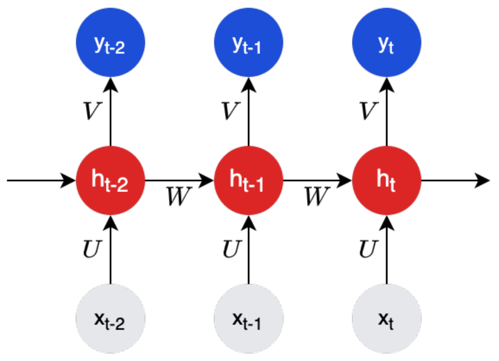


Figure 2.12: RNN architecture displayed in an *unrolled* way

and $\phi_2 = id$ we can denote the complete RNN cell by

$$\begin{aligned} a_t &= Wh_{t-1} + Ux_t + b, \\ h_t &= \tanh a_t, \\ y_t &= Vh_t + c. \end{aligned}$$

Similar to MLPs, RNNs also require backpropagation of the loss to update weights and minimize the empirical risk. In RNNs however, the output of one time-step depends on the previous ones. So the loss is *backpropagated through time* (BPTT) from the last time-step t to the first time-step, illustrated in Figure 2.14. In vanilla[1] RNNs, the hidden units only perform the non-linear activation with functions ϕ_1 and ϕ_2. However, the published literature on RNNs spawned several variations, such as the Long Short-Term Memory (LSTM) and the Gated Recurrent Unit (GRU) that perform more complex computations per hidden unit. Previous research has shown impressive performance for the classification and prediction of sequential data [33], [34].

[1]Not customized from its original form

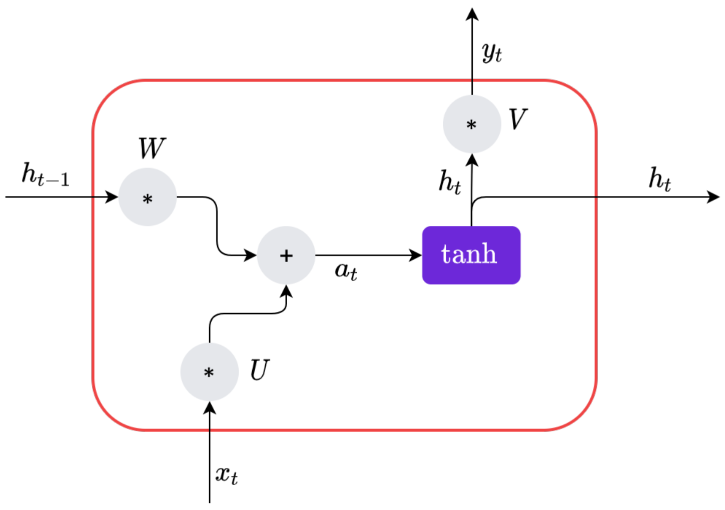


Figure 2.13: Vanilla RNN cell (biases are omitted for brevity)

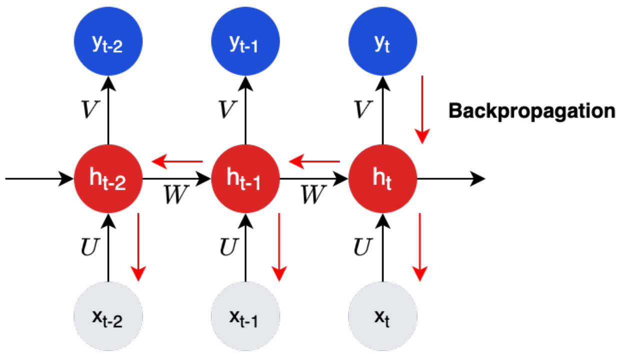


Figure 2.14: Backpropagation through time of an RNN

Long Short-Term Memory (LSTM)

Firstly, Long Short-Term Memory (LSTM) networks were introduced in 1997 and designed to overcome error back-flow problems in vanilla RNNs, such as vanishing and exploding gradients [33]. LSTMs are especially performant when the goal is to learn long-term dependencies in the data. In contrast to vanilla RNN cells, the LSTM cells have a more complex internal structure that includes a memory cell c_t with various gates to regulate the flow of information. In specific, the LSTM cell includes an input gate i_t, output gate o_t, and forget gate f_t while the memory cell can flow freely to the next hidden unit. A visual representation of the LSTM cell is given in Figure 2.15, where weight matrices W_f, W_i, W_c, and W_o maintain the weights for each gate separately. The σ symbol denotes the logistic sigmoid function in the following two diagrams.

LSTMs use the hidden state h_t as the output of the unit. To compute the output of the cell, the first step is to compute the output of the forget gate. The forget gate determines which past information is kept and which is forgotten by multiplying each value in the previous cell state c_{t-1} with vector values in the range $[0, 1]$.

$$f_t = \text{sigmoid}(W_f[h_{t-1}, x_t] + b_f)$$

Then, the add gate determines which new information should be memorized in the cell state.

$$c'_t = \tanh(W_c[h_{t-1}, x_t] + b_c)$$
$$i_t = \text{sigmoid}(W_i[h_{t-1}, x_t] + b_i)$$

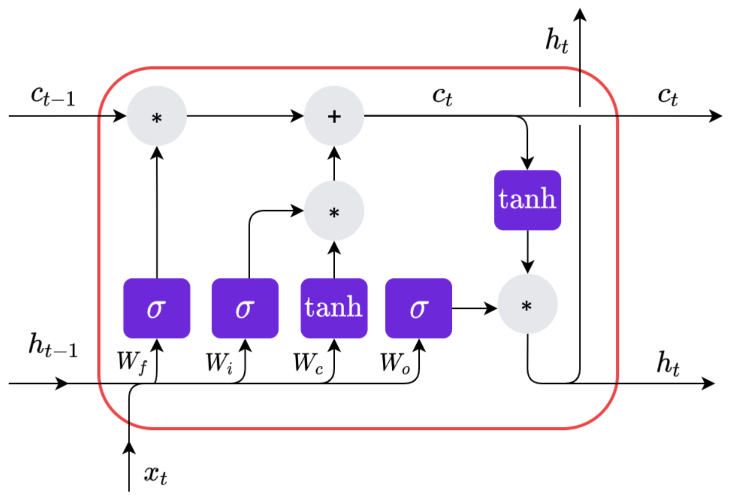


Figure 2.15: LSTM cell (biases are omitted for brevity)

Using this, the previous cell state is updated by

$$c_t = f_t \cdot c_{t-1} + i_t \cdot c_t'.$$

Finally, the output gate is used to calculate the next hidden state by

$$\begin{aligned} o_t &= \text{sigmoid}(W_o[h_{t-1}, x_t] + b_o) \\ h_t &= o_t \cdot \tanh(c_t) \end{aligned}$$

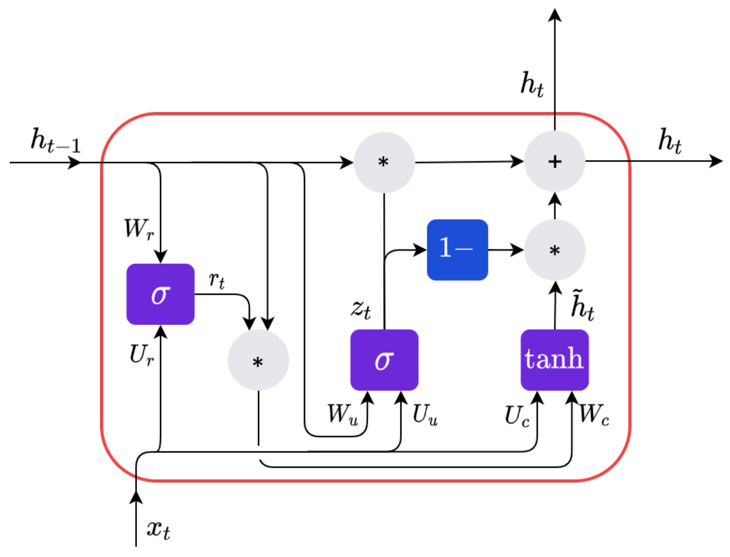


Figure 2.16: GRU cell (biases are omitted for brevity)

Gated Recurrent Unit (GRU)

Since the inception of LSTMs, other variations have been proposed. Arguably the most notable is the Gated Recurrent Unit (GRU). GRUs simplify the original LSTM cell by replacing the add, forget, and output gates with a reset r_t and update z_t gate [34]. With fewer parameters to train, GRUs

are computationally more efficient while achieving similar performance as LSTMs. The GRU cell is visualized in Figure 2.16, with weight matrices for each gate given by W_r, W_u, and W_c.
Computing the next hidden state is done as follows, which also serves as the output of the cell. First, the output of the gates are computed by

$$
\begin{aligned}
z_t &= \text{sigmoid}(W_u h_{t-1} + U_u x_t + b_u), \\
r_t &= \text{sigmoid}(W_r h_{t-1} + U_r x_t + b_u),
\end{aligned}
$$

allowing us to compute the proposed update through

$$\tilde{h} = \tanh(W_c(r_t \cdot h_{t_1}) + U_c x_t + b_c).$$

Lastly, the new value for the hidden state is calculated as

$$h_t = z_t \cdot h_{t-1} + \tilde{h}_t \cdot (1 - z_t).$$

Both LSTMs and GRUs have been successfully applied to tasks involving sequential data, ranging from language translation to image captioning [14]–[17].

For sequence to sequence modeling, an important advancement in deep learning has been the introduction of encoder-decoder RNN models [35], [36]. The encoder-decoder framework consists of a separate encoder and decoder, where the encoder is responsible for encoding an input sequence into a fixed-length vector and the decoder produces an output sequence from the encoded vector. Both the encoder and decoder are trained simultaneously on a data set to learn a mapping between input and output sequences.

Chapter 3

SQL Query Encoding

This chapter addresses the first research question and is concerned with the so-called *feature engineering* for machine learning of SQL queries. Feature engineering is the process of extracting features from raw data using domain knowledge, such that the features can be used by a predictive model. In our case, for any given SQL query we would like to extract its features such that the prediction of these features is *meaningful*. Within the context of relational databases, a meaningful prediction would mean that a correct prediction can be used to optimize some internal process and ultimately improve query efficiency and/or performance. Additionally, we also require the features to be of usable dimensions, such that they can be encoded into vectors and we avoid having too many parameters for our model.

The outline for this chapter is as follows. First, we discuss how parts of the query execution plans (QEP) represent intermediate results of the query. Then, we will discuss how the dimensionality of QEPs can be reduced using various heuristics. Subsequently, we introduce the concept of QEP fingerprinting, an encoding scheme that aims to capture the reusable elements of a query. The encoding scheme will then be formalized and presented together with with a motivation and an illustrative example. Lastly, algorithms will be presented for encoding raw SQL queries into vectors by means of QEP fingerprinting.

3.1 Deconstructing Query Execution Plans

Raw SQL queries, essentially strings of text, are highly dimensional and hard to represent in lower dimensions. Where current approaches rely on query templates as features for their predictive models, this work presents a novel method of encoding SQL queries based on the corresponding QEP. In contrast to raw SQL queries, the QEP provides a hierarchical structure (tree) representing the query without sacrificing information about the query. Additionally, compared to query templates, the encoding is more robust against small changes in the query and allows for *partially correct* predictions. As explained in Chapter 2, the QEP is defined by its operators, nodes that perform an action on their descendants. One can look at these operators in isolation and deduce (incomplete) information about the query, such as what tables are joined and what attributes are specified. Therefore, we propose to use the operators of QEPs as features, which leads us to introduce the concept of partial QEPs.

3.1.1 Partial Query Execution Plans

The notion of a partial QEP was first introduced by Marcus et al. [4] and used it to denote a forest of trees representing a QEP that is still being built. We take a slightly different approach, and consider a partial QEP to be any sub-tree of a QEP. Recall that the QEP represents the query as a sequence of steps to perform on the database to obtain the desired result. Furthermore, we have that for any sub-tree of the QEP, evaluating its defining operator — the root node of this sub-tree — depends solely on its child nodes and their descendants; i.e. changes in one branch of the query plan cannot affect any node outside of their ancestors. This implies that for any operation in the sequence of steps performed by the database management system (DBMS), the result of the operation depends only on the results from previous operations in the sequence. It follows that these operators can be considered independently and when evaluated yield some *intermediate result* of the query. In other words, any sub-tree of a valid QEP is also a valid QEP, which can be evaluated on the database to obtain some result. We define a partial QEP to be any sub-tree of a valid QEP. To demonstrate, consider the query in Algorithm 2, which aims to retrieve all the poster images of movies that are currently airing in cinemas.

Algorithm 2: Example SQL Query for poster images of movies currently airing in cinemas.

```
1: SELECT posters.image
2: FROM posters, movies
3: WHERE posters.movie_id = movies.id
4: AND movies.in_cinemas = true
```

Let p denote the QEP for this query, presented in Figure 3.1. One could write this QEP with notation[1] as

$$p = \Pi_{poster\_image}\Big(\Pi_{poster\_image,poster\_movie\_id}\big(\mathbf{T}(posters)\big) \bowtie_{\text{NL}} \Pi_{movie\_id}\big(\sigma_{movie\_in\_cinemas=true}\big(\mathbf{T}(movies)\big)\Big).$$

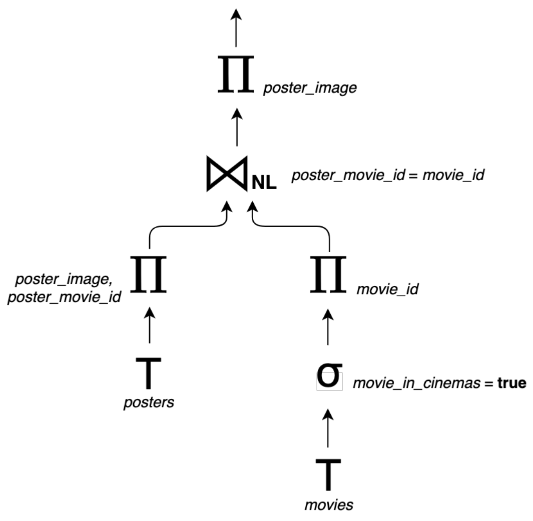


Figure 3.1: An example QEP for the query of Algorithm 2

The relation $\Pi_{movie\_id}\big(\sigma_{movie\_in\_cinemas=true}\big(\mathbf{T}(movies)\big)$ is a sub-tree of the QEP, and thus a partial QEP of p which can be evaluated independently. Let p' denote this partial QEP of p such that

$$p' = \Pi_{movie\_id}\big(\sigma_{movie\_in\_cinemas=true}\big(\mathbf{T}(movies)\big).$$

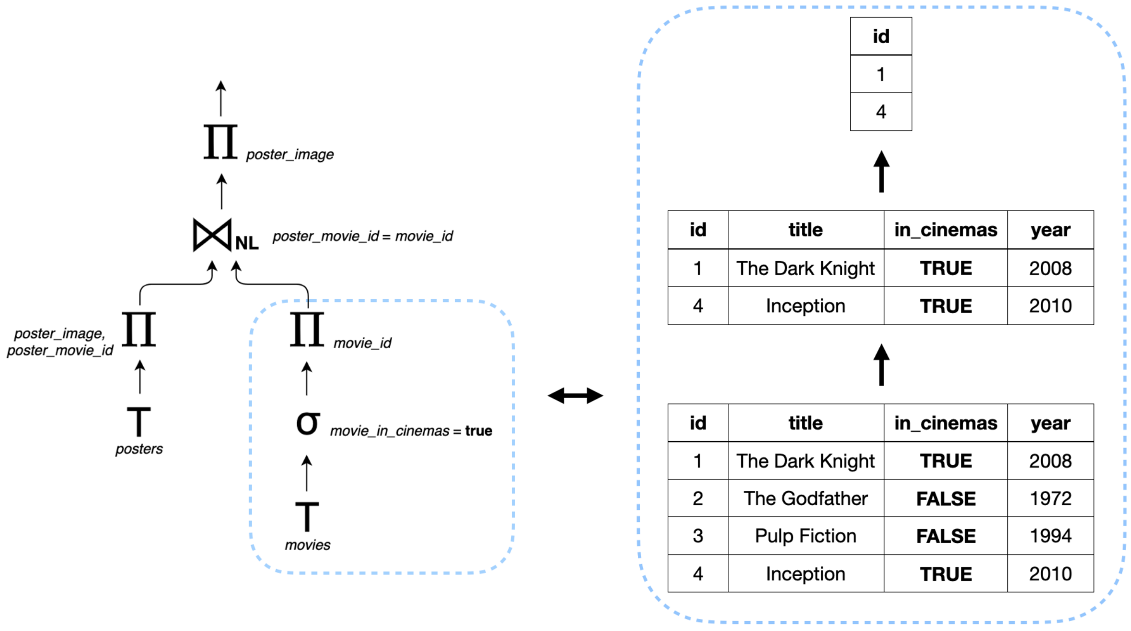
Naturally, we can represent the full QEP by substituting p' for this relation to obtain

$$p = \Pi_{poster\_image}\Big(\Pi_{poster\_image,poster\_movie\_id}\big(\mathbf{T}(posters)\big) \bowtie_{\text{NL}} p'\Big).$$

The notion of p' represents a relation that corresponds to an intermediate result of the query. Namely, p' represents the IDs of movies that are currently airing in cinemas. This relation can be

[1]Join attributes are omitted for brevity and **T** represents a table scan.

used as a substitute for the partial QEP p'. Hence, if this intermediate result was cached in memory, it could be used directly for evaluating p, without needing to re-evaluate p'. Naturally, it could be used for other queries that have p' as one of its partial QEPs, and the same can be said for every other partial QEP. Each operator in p' also corresponds to some intermediate result of the query. Figure 3.2 highlights p' in blue and shows the intermediate results of each operator in p'.


Π poster_image
⋈NL poster_movie_id = movie_id
poster_image, poster_movie_id Π
Π movie_id
T posters
σ movie_in_cinemas = **true**
T movies

id
1
4

id	title	in_cinemas	year
1	The Dark Knight	**TRUE**	2008
4	Inception	**TRUE**	2010

id	title	in_cinemas	year
1	The Dark Knight	**TRUE**	2008
2	The Godfather	**FALSE**	1972
3	Pulp Fiction	**FALSE**	1994
4	Inception	**TRUE**	2010

Figure 3.2: The partial QEP p' and the intermediate results of its operators

So, we can split any QEP corresponding to a query into a set of partial QEPs. These partial QEPs form a set of reusable elements that make up the QEP and can also re-occur in QEPs corresponding to other queries. For the example QEP in Figure 3.1, we identify the partial QEPs with a blue outline in Figure 3.3. In this case, we find that there are exactly 7 unique partial QEPs in the QEP.

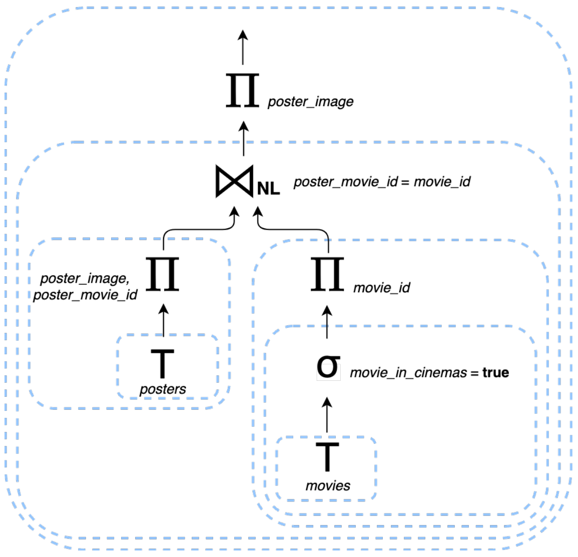

Π poster_image
⋈NL poster_movie_id = movie_id
poster_image, poster_movie_id Π
Π movie_id
T posters
σ movie_in_cinemas = **true**
T movies

Figure 3.3: Identified partial QEPs of the QEP in Figure 3.1

More formally, we let $\mathcal{S}(p) = \{p'_1, p'_2, \ldots, p'_n\}$ be the set of all n partial QEPs for a QEP p, with

$n \in \mathbb{N}^+$. I.e. we define a mapping

$$f : p \to \mathcal{S}(p) \tag{3.1}$$

that maps a QEP p to its set of partial QEPs $\mathcal{S}(p)$. The elements of $\mathcal{S}$ for the QEP in 3.1 could then be specified as

$$\begin{aligned}
\mathcal{S}(p) = \Big\{&\mathbf{T}(movies),\\
&\mathbf{T}(posters),\\
&\sigma_{movie\_in\_cinemas=true}\big(\mathbf{T}(movies)\big),\\
&\Pi_{poster\_image,poster\_movie\_id}\big(\mathbf{T}(posters)\big)\\
&\Pi_{movie\_id}\big(\sigma_{movie\_in\_cinemas=true}\big(\mathbf{T}(movies)\big),\\
&\Pi_{poster\_image,poster\_movie\_id}\big(\mathbf{T}(posters)\big) \bowtie_{\mathrm{NL}} \Pi_{movie\_id}\big(\sigma_{movie\_in\_cinemas=true}\big(\mathbf{T}(movies)\big),\\
&\Pi_{poster\_image}\Big(\Pi_{poster\_image,poster\_movie\_id}\big(\mathbf{T}(posters)\big) \bowtie_{\mathrm{NL}}\\
&\Pi_{movie\_id}\big(\sigma_{movie\_in\_cinemas=true}\big(\mathbf{T}(movies)\big)\big)\Big)\Big\}
\end{aligned} \tag{3.2}$$

3.1.2 Reducing Dimensionality

We have established the definition of a partial QEP to be any sub-tree of the QEP which represents some intermediate result of the query. The hierarchical structure of the QEP tells us that the result from evaluating a partial QEP can be used as a surrogate for that partial QEP. It is this dependency that allows us to dismantle a query into a set of partial QEPs, such that a correct prediction of a partial QEP can be used to optimize queries whose QEP includes the same partial QEP. However, the dimensionality of (partial) QEPs is still extremely large with a complexity of at least $n!$ where n is the number of tables in the database. This makes it difficult to vectorize QEPs without using some heuristics for reducing the dimensionality, especially if we consider *sets* of partial QEPs. To further reduce the dimensionality of the resulting vectors, we present several heuristics based on our observations.

Query Templates

Queries can be transformed into query templates before generating the QEP, similar to the approach taken for Apollo [5]. The main reason for this is that query parameters are represented by basic types, such as integers, floats or strings. Encoding these types is not practical given the size of their respective universes. For example, the simple query `SELECT * FROM cities WHERE name='Eindhoven'` already has infinitely many possibilities for the parameter `name`, given that it is a string. Hence, we transform queries into a query template by replacing all parameters with a placeholder. Thereby, the previous query would be transformed into `SELECT * FROM cities WHERE name='?'` and the queries `SELECT * FROM cities WHERE name='Eindhoven'` and `SELECT * FROM cities WHERE name='Rotterdam'` would both result in the same query template. The underlying structure of the query template is the same and still conveys information on what tables are accessed, which attributes are important, and what the result should look like.

Furthermore, observe that in some cases, filters can still be applied to the result of the query template. For example, the result from `SELECT * FROM cities` could still be filtered on `name='Eindhoven'` to obtain the result of the original query. These filter operations are much less costly to evaluate ($O(n)$) compared to compound operators such as joins ($O(n^2)$). Only when the result is projected on attributes not including the filter attribute, retroactive filtering becomes impossible without altering the QEP. Even then, altering the QEP to include the filter attributes in the projected attributes is trivial. To demonstrate, consider the example SQL query in Algorithm 2 with the corresponding QEP in Figure 3.1. Imagine the case where the filter for `movies.in_cinemas = **true**` was changed to `movies.in_cinemas = false` instead. Clearly, the intermediate results of partial QEP p' and its ascendant operators now differ as well, as it represents the movie IDs of movies *not* airing in cinemas. When queries are transformed into query templates, this differentiation is lost. However, all that is required is to slightly alter the QEP, by removing the filter operator, and including the filter attribute in any projection that is its ascendant node. Figure 3.4 shows how the QEP in Figure 3.1 could be altered to enable retroactive filtering for the `movies.in_cinemas` attribute.

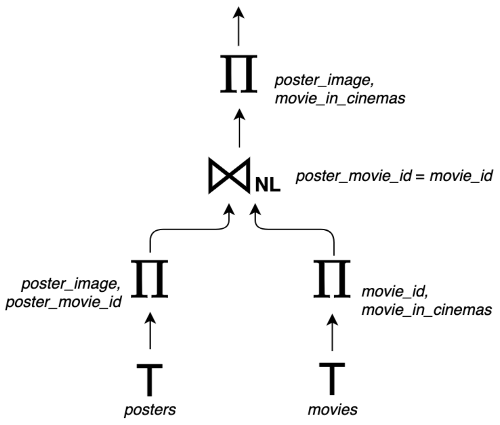

Π
poster_image, movie_in_cinemas
⋈NL
poster_movie_id = movie_id
poster_image, poster_movie_id
Π
Π
movie_id, movie_in_cinemas
T
posters
T
movies

Figure 3.4: The example QEP of Figure 3.1 altered to account for filter parameters

Moreover, the use of query templates improves the robustness of encodings against small changes. Queries whose results are highly similar but differ in their parameters would result in the same encoding. For example, consider two queries for products with `price > 100` and `price > 120`. These queries only differ in their result by items with a price between 100 and 120. Without this filter, the queries would yield the same result, and we could still obtain the result of the queries respectively by applying the filters afterward.

Operator Commutativity

An important thing to mention is that operators with more than one child node are not strictly commutative, such as antijoins[1], while others are, such as unions and inner joins. In the case of antijoins, a distinction can be made between the 'left' and 'right' tables of the join, and swapping these tables results in a different result. This implies that we cannot consider some arbitrary relation $\mathbf{T}(actors) \triangleright \mathbf{T}(movies)$ to be equivalent to $\mathbf{T}(movies) \triangleright \mathbf{T}(actors)$, and a distinction is also needed in the encoding.

Ideally, we would be able to capture the (non-)commutativity of operators in the encoding, such that changing the position of child nodes results in a different encoding for non-commutative operators while yielding the same encoding for commutative ones. However, we argue that assuming every internal operator to be strictly non-commutative has little impact on the dimensionality of the encoding. This is because the query optimizer is responsible for determining the optimal way of executing the query, so an inner join on the 'movies' and 'actors' tables will always result in the 'movies' table being the 'left' table for instance. The query optimizer is consistent in these delegations unless the contents of the subject tables change significantly.

Join Constraints

Also, note that only tables that share a key can be joined. For any two tables in a database, one or more shared keys is a prerequisite for being able to join the two tables. Therefore, instead of considering every unique combination of tables, we only have to consider the combinations that could

[1]The commonly used symbol for antijoins in relational algebra is $\triangleright$

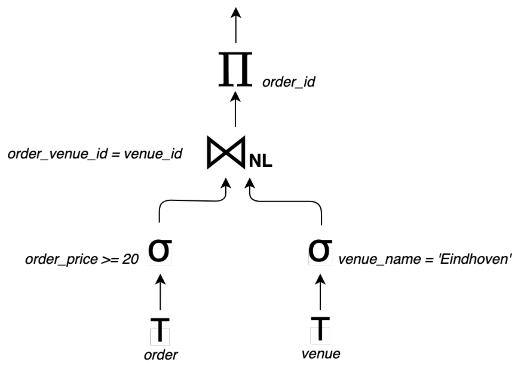


Figure 3.5: A QEP for the example SQL query in Algorithm 1

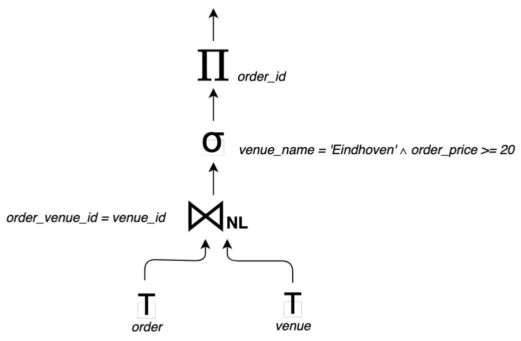

Π order_id
σ venue_name = 'Eindhoven' ∧ order_price >= 20
order_venue_id = venue_id
⋈NL
T order
T venue

Figure 3.6: An alternative QEP for the example SQL query in Algorithm 1

actually be encountered, and reduce the dimensionality without any loss of information. This implies that considering all $n!$ join orders in the encoding of a QEP is not needed, where n is the number of tables in the database, and we only have to consider those that are valid for the subject database.

Top-k Frequent Partial QEPs

After transforming each query into its corresponding query template, it is sensible to assume that the number of unique partial QEPs is drastically reduced. Experiments, that will be discussed later in Chapter 5, show that this assumption is very reasonable. However, in the case that the number of unique partial QEPs is still too high, we could only consider the e.g. top-5000 or top-10000 most frequent partial QEPs in our data set to encode. Even though the information could be lost on the less frequent partial QEPs, the dimensionality of the vectors will be directly limited by the chosen dictionary size.

Extension to Multiple QEPs

Recall that, for any given query q, the query optimizer of a DBMS might generate multiple QEPs, of which only the one with the lowest cost is eventually executed. For instance, for the SQL query in Algorithm 1 it was determined that the QEP in Figure 3.5 had the lowest cost, and would thus be the most optimal QEP to execute. However, an alternative QEP for this query is the one given in Figure 3.6. The query optimizer of a DBMS would have estimated its cost to be higher than the QEP in Figure 3.5, and as such it would not be executed. Nevertheless, it may be the case that the result of the partial QEP corresponding to $\mathbf{T}(order) \bowtie_{\text{NL}} \mathbf{T}(venue)$ was cached in memory. This is can happen if a previous, but different query was executed where this partial QEP was part of its most optimal QEP. Observe that in this case, executing the QEP in Figure 3.6 makes more sense, as the cached partial QEP can be used directly. In other words, the real cost of the QEP in Figure 3.6 is actually lower than that of Figure 3.5 when we account for cached partial QEPs. Hence, accounting for different QEPs per query could have a significant impact on the performance of any predictive model, and the extent to which a DBMS could be optimized.

To account for this, the encoding could include the top-k QEPs as estimated by the query optimizer. Consider an existing database and associated query log, a list of previously executed queries. This query log is an ordered sequence of queries and can be defined as $\mathcal{L}_q = \{q_1, q_2, ...q_n\}$, where n is the number of queries. Let $\mathcal{P}_k(q)$ denote the set of top-k most efficient QEPs for a query q with $k \in \mathbb{N}^+$, e.g. $\mathcal{P}_5(q)$ corresponds to the set of the top 5 most efficient QEPs generated for a query q. Thereby, we can define a mapping from q to p as

$$f : q \rightarrow p \in \mathcal{P}_k(q).$$

It follows that the query log $\mathcal{L}_q$ can be transformed into a sequence of sets of QEPs by mapping each query q to the set $\mathcal{P}(q)$, yielding $\mathcal{L}_p$

$$\mathcal{L}_P = \{p_{1_1}, p_{1_2}, \ldots, p_{1_k}\}, \{p_{2_1}, p_{2_2}, \ldots, p_{2_k}\}, \ldots \{p_{n_1}, p_{n_2}, \ldots, p_{n_k}\},$$

where n is the number of queries in the log and k is the number of QEPs considered per query.

Also, recall that each QEP p can be mapped to a set $\mathcal{S}(p)$ of its partial QEPs (sub-trees). Hence, mapping each $p \in \mathcal{P}_k(q)$ to its corresponding $\mathcal{S}(p)$ for each $\mathcal{P}_k(q) \in \mathcal{L}_q$ would result in a sequence of sets of sets. This structure is rather complex and no literature is known to exist that uses this structure in applied machine learning. Therefore, we propose two methods to flatten the sets of sets so the overall structure is simplified into a sequence of sets.

The first option is to consider all partial QEPs of each $p \in \mathcal{P}_k(q)$ as one set, i.e. take the union over all $\mathcal{S}(p)$

$$\bigcup_{p \in \mathcal{P}_k(q)} \mathcal{S}(p)$$

Note that in this case, we would still be modeling the temporal correlation between queries, as each QEP in $\mathcal{P}_k(q)$ yields the same result. In addition, the partial QEPs of every QEP in $\mathcal{P}_k(q)$ correspond to a valid QEP that can be executed. Therefore, predictions for the partial QEPs in subsequent queries remain valid, i.e. can be executed to obtain some intermediate result of the query. The downside is that the cardinality of the set $\mathcal{S}$ of a query could become much larger, up to k times as large, as we include many different partial QEPs.

Another approach is to consider only the most frequent partial QEPs for each $p \in \mathcal{P}_k(q)$. However, this would mean that the partial QEPs corresponding to the full QEP for each QEP in $\mathcal{P}_k(q)$ are rarely included in the encoding, given that these only occur once by definition. Since it would prevent predicting the QEP in full, it has a serious limitation.

Unfortunately, we were only able to obtain the top-1 QEP for queries during experiments. Hence, only one QEP will be considered per query in our experiments in Chapter 5. Experiments with multiple QEPs per query and comparing both possible approaches of handling them are left for future work.

Conditional Label Dependence

Lastly, there is a conditional label dependence between partial QEPs. In particular, some partial QEPs always imply the presence of other partial QEPs. For the query in Algorithm 2, and its set of partial QEPs in Equation 3.2, observe that the presence of $\Pi_{movie\_id}(\sigma_{movie\_in\_cinemas=true}(\mathbf{T}(movies))$ also implies the presence of $\sigma_{movie\_in\_cinemas=true}(\mathbf{T}(movies))$ in the QEP. Analogously, the presence of $\sigma_{movie\_in\_cinemas=true}(\mathbf{T}(movies))$ implies the presence of $\mathbf{T}(movies)$. In other words, for any given QEP, all its partial QEPs with more than one node imply the presence of other partial QEPs with one of its child nodes as its defining operator.

This dependency relation can be captured using a dependency graph, where nodes represent partial QEPs. However, instead of using edges to denote the *dependence* of one node on another, we use edges to denote which nodes are *implied* from other nodes. Hence, an edge from some node p'_1 to some node p'_2 means that when p'_1 is present, p'_2 must also be present in the set of partial QEPs. Figure 3.10 shows the graph of this toy example. Self-loops are implied, given that the presence of p'_1 always implies the presence of p'_1.

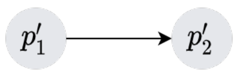


Figure 3.7: Dependency graph where the presence of p'_1 implies the presence of p'_2

Now assume we have some predictive model that predicts a set of partial QEPs $\mathcal{S}$ corresponding to a future query. Additionally, assume that $\sigma_{movie\_in\_cinemas=true}(\mathbf{T}(movies)) \in \mathcal{S}$ and $\mathbf{T}(movies) \notin \mathcal{S}$.

From theory, we can deduce that if $\sigma_{movie\_in\_cinemas=true}(\mathbf{T}(movies))$ is present, then $\mathbf{T}(movies)$ must also be present, so we know for certain that the model failed to predict one of the two partial QEPs. Since the model includes a measure of confidence in its predictions, i.e. it predicts the elements with the highest probabilities, the dependency graph can be used to deduce the implied elements from the model's top predictions. The inference of this graph can be done either as a post-processing step or built directly into the model. Chapter 5 will explain how both approaches work and compare their performance.

3.2 Query Execution Plan Fingerprinting

3.2.1 Overview

Based on the above, an encoding scheme was devised to transform a query log into a data set, QEP fingerprinting. Firstly, each query is transformed into a query template by replacing all filter parameters with a placeholder. Then, the query templates are executed on a dummy database that has an identical schema as the database that the query was originally executed on. Executing the queries allows us to obtain the QEP of the query. Subsequently, the QEP is recursively split into the set of distinct partial QEPs. Each partial QEP is then hashed with a hash function[1] to combine the information of the operators and the structure of the partial QEP into a single representation, i.e. a fingerprint. Any change in the partial QEP, such as an alternate join order, would result in a different hash. Changes in filter parameters do *not* result in a different hash, since they were replaced with placeholders in the QEP as per Section 3.1.2. Thereby, the set of partial QEPs is transformed into a set of hashes. Afterward, each hash of the top-N most frequent partial QEPs is assigned a unique integer index from the set $\{1, \dots, N\}$, where N denotes the dictionary size to be used. The dictionary size can be equal to the number of unique partial QEPs, so as to consider every partial QEP. Alternatively, it can be limited to a fixed number, such as 5000, as was described in Section 3.1.2. During the process, we build the dependency graph to maintain a mapping between the partial QEPs and the partial QEPs it implies. Lastly, each set of hashes is transformed into a set of indices, where hashes of partial QEPs not in the top-N most frequent partial QEPs are discarded. The indices are used to write a 1 to the output vector at the corresponding index to indicate the presence of a partial QEP. In simpler terms, we assign a unique index to each unique partial QEP we encounter in the log after transforming queries into query templates.

3.2.2 Rationale

This section briefly outlines the reasoning for the proposed encoding scheme.

First of all, every partial QEP can be evaluated independently and substituted with the result of that partial QEP, making partial QEPs meaningful to predict. The prediction of partial QEPs enables several forms of optimization, such as selective caching, where predictions can be used to decide which results of one or more partial QEPs should remain in memory. The cached results could be used directly in evaluating QEPs that contain the same partial QEPs and makes re-evaluating these partial QEPs unnecessary. Alternatively, if the model predicts some partial QEPs to occur in the next sequence of queries, a DBMS could pre-compute these partial QEPs and keep their results cached. Either case would improve query efficiency by reducing computation time and I/O operations, as some intermediate results of the query are already in memory. In distributed systems, the results of partial QEPs could also be sent to edge nodes. Consider a QEP with a join on two arbitrary relations for example. If one of the relations was predicted and its result sent to an edge node of the network, it would only have to send the other relation at the time of the query. This would reduce the amount of data to be sent over the network at the time of the query and consequently improve query speed.

[1]The type of hash function is irrelevant as long as collisions are avoided. In experiments, we use the default hash function of Scala's hash table implementation. However, alternative hash functions can also be used.

In addition, hashing captures all the information of a partial QEP into a small integer representation and has several benefits. Firstly, the hash represents the intermediate result of the partial QEP. Any change of the partial QEP that alters the result also alters its associated hash value. The dependency relation between partial QEPs is also captured as a dependency relation between their encoded integer representations. As such, a dependency graph can be used to efficiently model the implied partial QEPs for any given partial QEP. Secondly, since only partial QEPs that were previously seen are considered, all predictions correspond to valid partial QEPs that can be executed. For example, the model can only predict join operators on tables that share a key and can actually be joined. This also alleviates the need to explicitly determine which joins to consider in the encoding as per Section 3.1.2. But, it comes at the cost of not being able to predict partial QEPs that were previously unseen. Thirdly, the dimensionality of the encoding is reduced to the number of unique partial QEPs present in the training data. If needed, the number of considered partial QEPs could be limited to only the top-most frequent partial QEPs.

Furthermore, encoding a query into a set of re-usable elements gives us the ability to make predictions that are useful even when only partially correct. For example, if a model only manages to predict some elements of $\mathcal{S}(p)$ correctly for some future query, these predictions can still be used to improve query performance. So, instead of predicting each subsequent query (in)correctly in its entirety, the model can predict each subsequent query to the best of its ability. This also means that we can directly model the relationship between partial QEPs occurring in the past and those occurring in the future. Take for instance a sequence of 10 distinct queries. Even though the queries are distinct, they may all access the same particular table, such that their set representations all include the same partial QEP.

Also, the representation is robust against changes that affect higher-level operators, the root node of the QEP, and the nodes close to it. For example, adding an `ORDER BY` clause to the query only alters the order of the results. Generally, this only results in the addition of an ordering operator as the root node of the QEP. If we break this QEP into the set of its partial QEPs, we find that the set representation remains the same except for one additional partial QEP corresponding to the full QEP. Hence, most of the representation remains the same.

Moreover, the representation is flexible in a way that certain attributes can be included or excluded from the encoding for the level of detail that is required given the dimensionality constraints. On one hand, if it was known that there are only a small number of different filter parameters, the step of transforming queries into query templates could be skipped such that the filter parameters are included in the representation. In this case, a different filter parameter would result in a different hash value, and thus a different integer representation. For instance, the queries `SELECT * FROM cities WHERE name='eindhoven'` and `SELECT * FROM cities WHERE name='rotterdam'` would have a different encoding. On the other hand, if the goal was to only predict the tables and indices involved in the query, one could only consider leaf nodes of the QEP in the set of partial QEPs $\mathcal{S}$.

Lastly, it would integrate well with existing query optimizers in DBMSes. When a DBMS tries to determine the optimal QEP for a query, it might generate multiple plans with an associated cost of which only the QEP with the lowest cost is eventually executed. However, the query optimizer could now ignore the cost of the partial QEPs that were predicted and cached in memory, and determine the most optimal QEP taking that into account. In theory, it could generate QEPs such that the use of the predicted relations is maximized.

3.2.3 An Illustrative Example

We demonstrate the encoding scheme using the example QEP in Figure 3.1. First, the QEP is split into the set of partial QEPs $\mathcal{S}$ as specified in Equation 3.2 and Figure 3.3. Each partial QEP in $\mathcal{S}$ is hashed to obtain a set of hashes, which are then assigned to a unique index. The indexed partial QEPs are shown in Figure 3.8. Thereby, the set of indices resulting from the encoding of this QEP is given by $\{1, 2, 3, 4, 5, 6, 7\}$. Note that every index corresponds to the hash of a partial QEP.

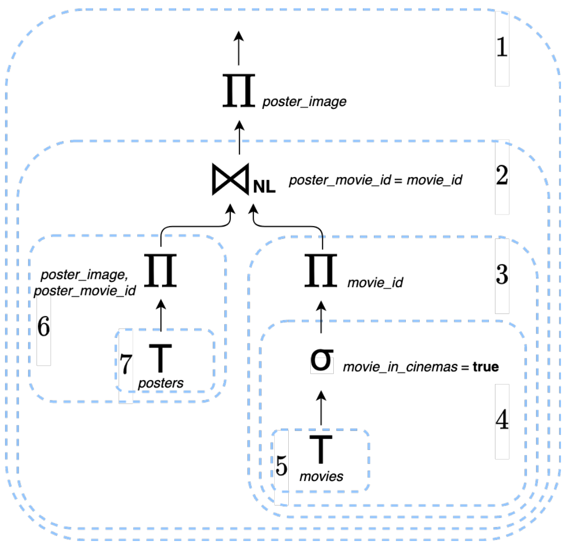


Figure 3.8: Indexed partial QEPs of the example QEP in Figure 3.1

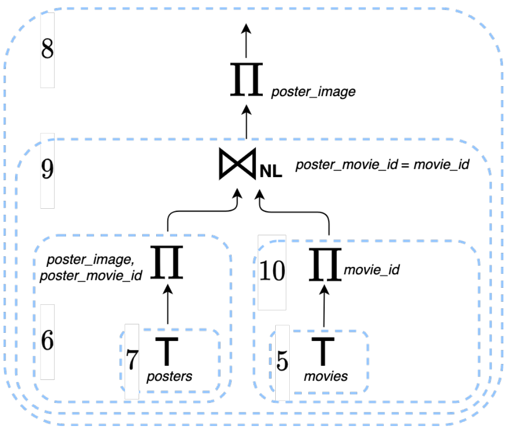
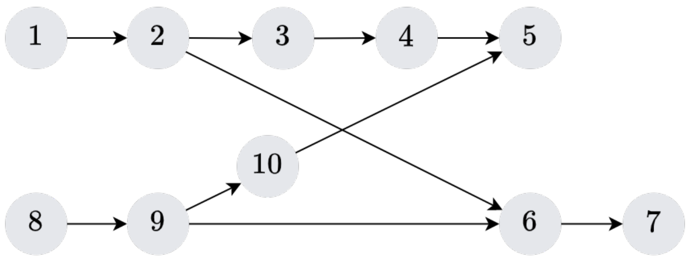


Figure 3.9: Indexed partial QEPs of another query

If we now consider a similar, but different query that drops the requirement for the movies currently airing in cinemas, we could obtain the plan shown in Figure 3.9. The set of indices from encoding this plan is given by $\{5, 6, 7, 8, 9, 10\}$. Observe that three unique partial QEPs occur in both plans, and the encoding captures this information by including the indices 5, 6, and 7 in both encodings. Also observe that the higher level QEPs have different indices because the filter for `movie_in_cinemas` is not present in the QEP of figure 3.9. Since the filter is not present, the intermediate result of partial QEP 10 is different[1] from 3. The same can be said for plans 2 and 9, and 1 and 8. Thus, a change in a lower level of the QEP results in a different index for all higher-level partial QEPs, due to the change in hash.

We build the dependency graph for these encoded plans to obtain the graph in Figure 3.10. For each indexed partial QEP, we create a node corresponding to its index. Subsequently, we add directed edges between the nodes to model the relation between partial QEPs. Using this graph, we can find all the partial QEPs that are implied for any partial QEP. For example, it can be seen that the partial QEP indexed at 9 always implies the presence of partial QEPs 6 and 10, which in turn imply the presence of 5 and 7.



$$\begin{tikzcd}
1 \arrow[r] & 2 \arrow[r] \arrow[drrr] & 3 \arrow[r] & 4 \arrow[r] & 5 & \\
& & 10 \arrow[urr] & & & \\
8 \arrow[r] & 9 \arrow[ur] \arrow[rrr] & & & 6 \arrow[r] & 7
\end{tikzcd}$$

Figure 3.10: Dependency Graph

The algorithm for the encoding scheme is presented in Algorithm 3. The subroutine `getPartialQEPs`, which corresponds to the mapping in Equation 3.1, is listed in Algorithm 4.

[1]The root operator of partial QEP 3 is a projection on `movie_id` so applying the filter of partial QEP 4 afterward is not possible, hence why this operator is positioned here.

Algorithm 3: Query Encoding

```
1: **Input:** query log $\mathcal{L}$, database schema $S$, dictionary size $N$
2: **Initialize:** dummy database $D$ with schema $S$, encoded queries $X \leftarrow []$,
   dependency graph $G \leftarrow \{\}$
3: **for** each query $q$ in $\mathcal{L}$ **do**
4:   $\mathbf{v} \leftarrow [0, 0, \ldots, 0]^N$ ;              // Binary vector of encoded query
5:   $q' \leftarrow \text{template}(q)$ ;                      // Transform into query template
6:   $p^* \leftarrow \text{QEP}(q', D)$ ;                      // Evaluate query template to obtain QEP
7:   $\mathcal{S} \leftarrow \text{getPartialQEPs}(p^*)$ ;     // Recursively get partial QEPs of a QEP
8:   **for** each partial QEP $p$ in $\mathcal{S}$ **do**
9:     $G[i] \leftarrow \text{getPartialQEPs}(p)$ ;           // Add edge to implicit elements
10:    $h_p \leftarrow \text{hash}(p)$ ;                      // Hash function
11:    $i \leftarrow \text{mod}(h_p, N)$ ;                    // Convert to index
12:    $\mathbf{v}[i] \leftarrow 1$ ;                         // Write 1 at index
13:  **end for**
14:  $X.\text{append}(\mathbf{v})$
15: **end for**
16: **return** encoded queries $X$
```

Algorithm 4: getPartialQEPs

```
1: **Input:** a query execution plan $p$
2: **Initialize:** set of partial query execution plans $\mathcal{S} \leftarrow \{\}$
3: $\mathcal{S}.\text{add}(p)$
4: **if** $p.\text{children} \neq \{\}$ **then**
5:   **for** each child $p'$ in $p.\text{children}$ **do**
6:     $\mathcal{S} \leftarrow \mathcal{S} \cup \text{getPartialQEPs}(p')$ ;   // Recursive call
7:   **end for**
8: **end if**
9: **return** set of partial QEPs $\mathcal{S}$
```

Chapter 4

Prediction with Deep Learning

This chapter addresses the second research question and is concerned with creating a deep learning model for the encoded query data. The previous chapter introduced an encoding scheme that transforms the query log into a sequence of sets. Therefore, given a sequence of sets, we require a model of the joint distribution of the next sequence of sets.

This chapter introduces a model architecture suitable for sequence to sequence modeling based on research of a similar problem. Subsequently, the model architecture is described and key components of the model are addressed. Lastly, it is explained how the model can be used to make predictions, and algorithms for inference are presented.

4.1 Model Architecture

As the SQL queries have been encoded as described in the previous chapter, the resulting data structure is a sequence of sets. However, vanilla RNNs only directly define the probability for sequences, and machine learning research that caters to sets is limited. Previous research investigated the use of sets as if they were sequences, such that each individual element of the set is fed into a corresponding RNN unit. They showed that the order in which the elements are read has an impact on the performance, despite there being no ordinal relation between them [37]. Unfortunately, their work focuses on sets to sets modeling and does not address working with sequential sets.

Perhaps most applicable to our problem is research published on solving the sequential sets to sequential sets learning problem [38]. One could draw parallels between our problem and theirs: Given the past orders of customers, can we predict the items that are likely to be bought in the future? To this end, they propose their own encoder-decoder framework. The encoder is responsible for encoding each sequential set into a lower dimension, this is done by learning a set embedding jointly with the input to output sequence mapping. Furthermore, they leverage a set-based attention mechanism to model the set-element interactions (from past set to future element), and explicitly model the element-element relation (from past element to future element) with an auxiliary vector γ that maintains the probabilities of elements appearing in the past sets. The decoder is responsible for decoding the sequence as well as the elements in each set. As their method seems applicable to our problem, it will serve as the foundation of our approach.

Since our problem revolves around finding correlations in temporal sets, we base our implementation on the encoder-decoder framework with an attention mechanism and its extension to sequential sets [16], [38]. Let us formalize the problem: given a sequence of sets of partial query execution plans (QEPs) $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_l\}$, where l denotes the length of the input sequence and $\mathbf{v}_i$ the vector containing the partial QEPs appearing in the i-th query, predict the subsequent m sets of partial QEPs $\{\hat{\mathbf{v}}_{l+1}, \hat{\mathbf{v}}_{l+2}, \ldots, \hat{\mathbf{v}}_{l+m}\}$.

A visual representation of the proposed framework can be seen in Figure 4.1. First, each set of partial QEPs $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_l\}$ is embedded into a lower dimension, as will be described in more detail later. The embedded sets $\{x_1, x_2, \ldots, x_l\}$ are fed into their corresponding RNN units to obtain the hidden state vector h by

$$h_t = f(x_t, h_{t-1})$$

where f is the combination of a GRU and an embedding function and t indicates the time step. The hidden state of the last GRU h_l in the encoder is used as the first hidden state s_0 of the decoder. The decoder also consists of RNN units, that take as input the previous hidden state s_{t-1}, the previous output y_{t-1}, and a context vector c_t which is obtained from the attention mechanism that will be explained later. The next hidden state in the decoder is given by

$$s_t = g(y_{t-1}, s_{t-1}, c_t),$$

where g is also the combination of a GRU and an embedding function, and y_{t-1} is the set embedding of $\hat{\mathbf{v}}_{t-1}$. Finally, we can compute the output using a softmax function

$$o(\hat{\mathbf{v}}_t) = \text{softmax}(W_o s_t) \tag{4.1}$$

where the hidden state is projected back to the space of elements with weight matrix W_o.

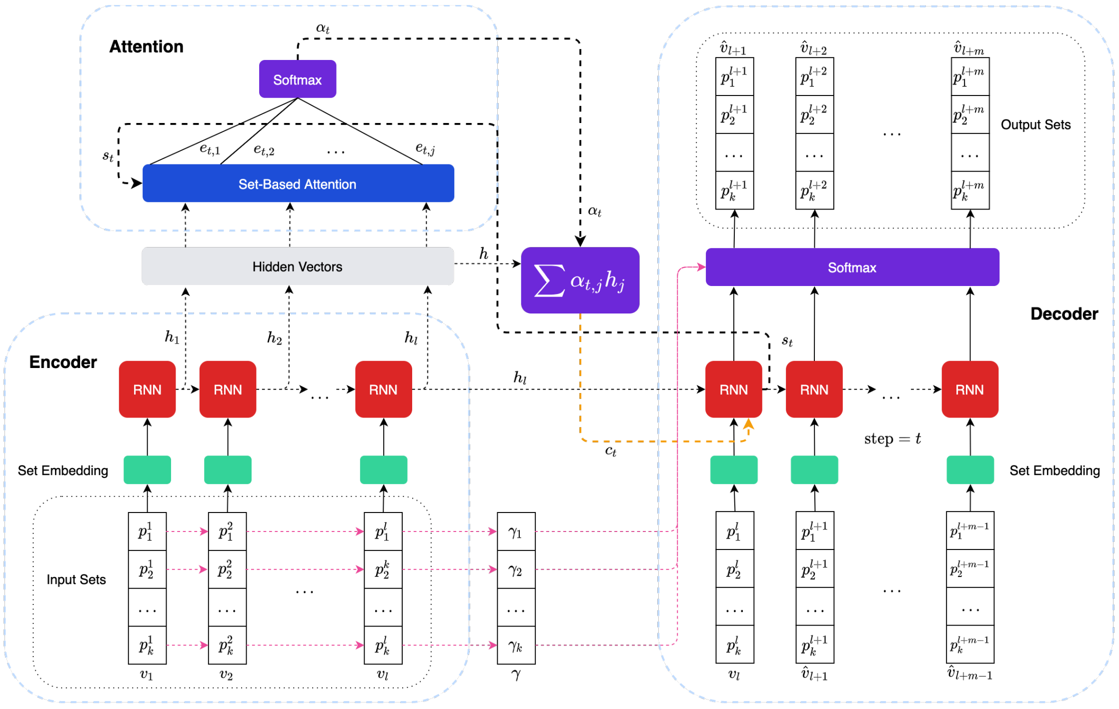


Figure 4.1: Model architecture of the encoder-decoder RNN framework.

4.1.1 Set Embedding

From the encoding scheme, we know there are N possible partial QEP in our encoding. As a result, the encoded queries have a dimension of N, where a 1 indicates the presence of some partial QEP in the set. These vectors are sparse, which makes models hard to train due to the large number of parameters that need to be learned. Hence, an embedding layer is used to compress the input feature vectors into vectors of a smaller dimension. This is done using an embedding matrix that is trained in the model directly. A detailed diagram of the set embedding layer that is also visible in Figure 4.1 is shown in Figure 4.2. First, the input sets are transformed into one-hot encoded vectors corresponding to each element in the set. Subsequently, an element matrix is used to map the one-hot encoded element to the corresponding dense representation. These element embeddings are then aggregated by average to obtain the final set embedding, which is then forwarded to the corresponding RNN unit. In a sense, the set embeddings are a low-dimensional representation of the query.

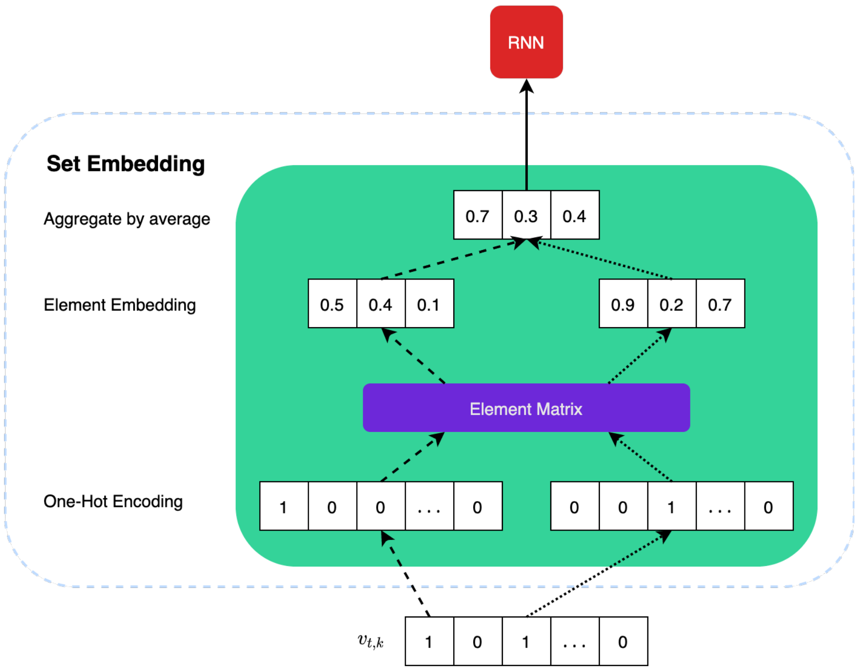


Figure 4.2: Set embedding layer for embedding input sets into a lower dimension and forwarding to an RNN unit.

4.1.2 Set-based Attention

It has been argued that sequence to sequence modeling can be difficult with encoder-decoder architectures [16]. As the entire sequence is compressed into a single vector, performance suffers especially in the case of longer sequences. To address this potential issue, we use an *attention mechanism* to use the input sequence as a context and allow the decoder to look back in time by giving it access to the input data [16].

At each time step t, the context vector c_t is provided as input to the GRU cells in the decoder. As a result, the decoder has access to the encoded sequence as well as the individual elements of the input. This allows the model to be more efficient and ideally also better at modeling the conditional probability $\mathbb{P}(y_1, \ldots, y_m \mid x_1, \ldots, x_l)$. The attention vector α, indexed by the time step t, is used to *attend* to different input elements with different weights at each time step t. The context vector c_t is computed as follows. First, the logits[1] e_{tj} are computed from the alignment model a such that

$$e_{tj} = a(s_{t-1}, h_j).$$

The alignment model holds the weights for scoring the match between inputs around position j and the output at time step t. Scores are calculated from the hidden state s_{t-1} and the j-th hidden state h_j of the input sequence [16]. This alignment model is implemented as an MLP which is jointly trained with the rest of the model. The attention vector is calculated for each hidden state h_j by

$$\alpha_{tj} = \frac{\exp(e_{tj})}{\sum_k \exp(e_{tk})}.$$

[1]The vector of non-normalized outputs of a predictive model, generally they form an input to the softmax function to obtain a vector of normalized probabilities.

Lastly, we compute the context vector c_t with the weighted sum of all the hidden states

$$c_t = \sum_j \alpha_{t,j} h_j.$$

In summary, we are able to model the conditional probability through the output of the GRU with Equation 4.2, where $\mathbf{x} = \{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_l\}$.

$$o(\hat{\mathbf{v}}_t) = \text{softmax}(W_o s_t) = \mathbb{P}(y_t \mid \{y_1, \ldots, y_{t-1}\}, \mathbf{x}) \tag{4.2}$$

Using the chain rule, we can now model the probability over the output sequence as

$$\mathbb{P}(\mathbf{y} \mid \mathbf{x}) = \prod_{t=1}^{m} \mathbb{P}(y_t \mid \{y_1, \ldots, y_{t-1}\}, \mathbf{x}),$$

where $\mathbf{x} = \{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_l\}$ and $\mathbf{y} = \{\mathbf{v}_{l+1}, \mathbf{v}_{l+2}, \ldots, \mathbf{v}_{l+m}\}$. The application of the chain rule makes this approach of modelling the joint conditional probability assumption free [37].

4.1.3 Modeling Repeated Elements

Research by Hu and He also include a method to specifically model the finer grain past element to future element relation [38]. In particular, they argue that elements occurring in the past are more likely to occur in the future. In our case, it is sensible to assume that partial QEPs frequently occurring in past queries are also likely to reoccur in future queries. Experiments in Chapter 5 will test the effect of this component and determine if this assumption holds.

Hence, they introduce a vector γ which contains the probability of the corresponding item appearing in the past set. Vector γ can also be seen in the model architecture in Figure 4.1. We implement vector γ to contain the probability of a partial QEP appearing in the past training data. Consequently, we change Equation 4.1 to

$$o(\hat{\mathbf{v}}_t) = \text{softmax}(W_o s_t \circ (\mathbf{1} - \beta \circ \mathbf{w}) + \gamma \circ \mathbf{w}) \tag{4.3}$$

with $\circ$ denoting the Hadamard product, vector β a binary vector maintaining the non-zero entries of γ, and a weight vector $\mathbf{w}$ to balance contributions from s_t and γ. Vector $\mathbf{w}$ is calculated as

$$\mathbf{w} = \text{sigmoid}(W_\gamma \gamma + \mathbf{z}),$$

with weight matrix W_γ and a vector $\mathbf{z}$ learned from the data. The purpose of β is to only have contributions from vector γ for partial QEPs which have appeared in the past.

4.2 Model Inference

Using the trained model we also need to make predictions. When provided with an input sequence of length l, it outputs the probabilities of the next sequences with length m. The sequence with the highest probability is selected as a prediction such that

$$\underset{y_1, y_2, \ldots, y_m}{\arg\max}\, \mathbb{P}[\{y_1, y_2, \ldots, y_m\} \mid \{x_1, x_2, \ldots, x_l\}].$$

To determine the elements in each set we use a greedy algorithm that incorporates the conditional label dependency between elements described in Section 3.1.2. Algorithm 5 predicts the elements with the top-k highest probability for each set. The output vector of size k is then created by setting the values in $o(\hat{\mathbf{v}}_t)$ that are the top-k most likely to 1 and the rest to 0. Furthermore, for each of these items, we know that their presence indicates the presence of other elements, which may not be directly predicted. Thus, we can use the dependency graph to exploit the label dependency and also set the implicit elements in $o(\hat{\mathbf{v}}_t)$ to 1.

There are two possible ways of using the dependency graph. The first option is to essentially use the dependency graph as a post-processing step, and not during model training, i.e. don't feed the implied elements into the next recurrent unit. The reason for this is that when predicting the next sets recurrently, the dependency graph could introduce more errors into the input of the next time-step. As there are always errors in each prediction, the dependency graph could compound on these errors which could affect the training of the model. However, the dependency graph could also be incorporated directly into the model, such that the elements implied by the dependency graph are combined with the predictions to form the input of the next recurrent unit. Experiments in Chapter 5 compare both these methods of using the dependency graph.

The algorithm for greedily decoding the next m sets is given in Algorithm 5. To incorporate the dependency graph into the model, line 12 can be moved after line 16, such that the decoder input includes the implicit elements found through the dependency graph.

Algorithm 5: Greedily decoding the next m sets

```
 1: Input: input sequence $\mathbf{x}$, number of set elements $k$, length of output sequence $m$, dependency
    graph $G$, $Encoder$, $Decoder$
 2: $\mathbf{y} \leftarrow$ [] with length $m$ ;                 // Output sequence
 3: $h \leftarrow Encoder$.initHidden() ;              // Hidden state of the Encoder
 4: for each input set $\mathbf{v}$ in $\mathbf{x}$ do
 5:    encoder_outputs, $h \leftarrow Encoder(\mathbf{v}, h)$
 6: end for
 7: decoder_input $\leftarrow$ last set of $\mathbf{x}$
 8: $s \leftarrow h$ ;                                   // Hidden state of the Decoder
 9: for each step $t = 1$ to $m$ do
10:    decoder_output, $s \leftarrow Decoder$(decoder_input, encoder_outputs, $s$)
11:    $\hat{\mathbf{v}}_t \leftarrow$ decoder_output.getTopK($k$) ;      // Top-$k$ highest probability
12:    decoder_input $\leftarrow \hat{\mathbf{v}}_t$
13:    for each element $p$ in $\hat{\mathbf{v}}_t$ do
14:       implicit_elements $\leftarrow$ getImplicitElements($G$, $p$)
15:       $\hat{\mathbf{v}}_t \leftarrow \hat{\mathbf{v}}_t \cup$ implicit_elements
16:    end for
17:    $\mathbf{y}$.append($\hat{\mathbf{v}}_t$)
18: end for
19: return $\mathbf{y}$
```

Chapter 5

Experiments

Chapter 3 introduced an encoding scheme for the feature engineering of SQL query data and Chapter 4 introduced a deep learning architecture for creating a predictive model for this type of data. This chapter builds upon the theoretical work in these chapters by implementing both the encoding framework and predictive model. The goal of these experiments is to measure the performance of the model and the effect of several of its components, such as the dependency graph. In addition, experiments are performed with the model in a more practical setting to get an indication of the model's usefulness in a database management system (DBMS). What follows is a series of experiments on the query logs of two subject databases. First, the raw SQL query logs were encoded using the encoding framework and a frequency analysis clarifies the underlying distribution of the data sets. Subsequently, the model was trained using different parameter settings and evaluated using several methods and metrics. Lastly, the computation time of the encoding framework, model training, and model inference are briefly addressed.

5.1 Collected Data

For our experiments, we require a database query log with an accompanying database schema. Since the goal is to exploit query patterns, it is crucial that we don't use synthetic data sets, but work with real data instead. Most public database benchmarks are intended to measure the internal performance of a DBMS [39]. As such, they are artificially generated to provide a synthetic workload sampled from known distributions. Additionally, their order and frequency of execution can be defined by the user. Some notable examples are TPC-H for analytical workloads and TPC-C for transactional ones [40]. Unfortunately, the amount of publicly available query logs is also limited. This is likely because query logs tend to contain sensitive information and publicizing them poses a security risk. The same can be said for the database schema. So, there is little incentive for people to share this information. To our knowledge, there is one publicly available data set of non-synthetic queries spanning a reasonable time frame[1]. This is the PhoneLabs data set published in 2016 [42], which recorded the embedded database of Android smartphones and anonymized the query logs before publication. In addition, we created our own data set by logging the SQL statements of a database supporting an Internet of Things (IoT) platform. Thus, both data sets are based on real-world data and have a sequential order of queries that is correct.

[1]We are aware of the SDSS SkyServer public database [41], which is not a published data set but has publicly accessible query logs. However, many queries could not be run in our experiments due to schema, function and view definitions specific only to their database. As a result, too many queries could not be encoded and it was deemed impractical to use.

5.1.1 Internet of Things (IoT)

The IoT data set was created through a partner company by logging the SQL statements of a PostgreSQL database serving as the DBMS of a medium-sized IoT platform. The platform is used by 500+ active users and allows users to manage and track IoT devices. It includes support for user management, device registration, device (fleet) management, over-the-air firmware updates, live sensor readouts, eSIM management, mobile data plans, and more. The backend of the platform uses the object-relational mapping (ORM) software Sequelize to generate SQL statements from a Node.js javascript runtime.

The data set consists of a two-month-long trace of all SQL statements executed on the PostgreSQL instance from April 15th through June 15th, 2021. In total, we collected 169689 queries, performing CRUD operations on exactly 20 tables. The schema was unchanged for the duration of the data collection. Logging was enabled by passing the `logging_collector=on` and `log_statement=all` command-line arguments to the PostgreSQL instance.

5.1.2 PhoneLabs

The second data set that we use for our experiments is the PhoneLabs dataset [42]. Smartphones running the Android operating system have a persisting embedded database to share private data between system operations and apps. In Android 4.4.4 (KitKat), this embedded database uses SQLite internally and provides various interfaces for processes to use. For a handful of subject smartphones, all device activity was recorded for 30 days while the phone was in normal use. The PhoneLabs dataset also includes the query log of the embedded SQLite databases and provides the full one-month trace of SQLite activity on a per phone basis. To protect the users' privacy, the query log was anonymized before publication by replacing query parameters with placeholders, similar to the process of creating query templates.

In contrast to the IoT dataset, the embedded database in mobile phones is subject to queries that are generated by an operating system rather than a single application. The operating system is host to a variety of applications, which each have its own use cases. Applications can access tables and even create their own tables in this database. This results in a very high amount of tables inside the database, with many of them being unused. Experiments will focus on one randomly selected subject smartphone, the device with the ID `2747d54967a32fc95945671b930d57c1d5a9ac02`. For this smartphone, we found 808 unique schema-defining messages in the database log. This data set also experienced a much higher volume of queries than the IoT data set, so for practical reasons we only use a subset of the first 5 days worth of data, which corresponds to 705668 recorded queries.

5.1.3 Data Cleaning

Several steps were performed to transform the raw data into a usable data set. For our experiments, we will only focus on `SELECT` queries, which aim to retrieve data from the database.

IoT

For the IoT query log, the raw data contains all logging statements written by the PostgreSQL instance. Therefore, we filter the raw data by excluding log messages not containing database query statements. Given that we only focus on `SELECT` queries, we exclude the queries not containing a `SELECT` query statement as well. Additionally, we exclude queries for the tables prefixed by 'pg', which correspond to tables used internally by PostgreSQL of which the schema is unknown. From the raw data, we extracted a total of 169689 queries, of which there were 127958 `SELECT` queries. The full frequency distribution of query types can be seen in Table 5.1.

Query parameters contain sensitive information such as names, addresses, and payment details. Thus, to preserve the privacy of the platform users[1], query parameters had to be replaced with placeholders. We created a script for replacing query parameters with placeholders based on a series of *regular expressions*. These regular expressions were used to target a number of clauses containing query parameters that we identified in order to replace them with placeholders. These clauses are listed below:

- Elements following a boolean operator, e.g. `=?, <?, >?, <=?, >=?`
- Elements within brackets following an `IN` keyword, e.g. `IN (?, ?, ?, ?)`
- Elements denoting a range following a `BETWEEN` keyword and surrounding an `AND` keyword, e.g. `BETWEEN ? AND ?`

Note that the anonymization of query logs is analogous to the creation of query templates as described in Section 3.1.2. The full schema of the database was obtained from the PostgreSQL instance and exported in CSV format. The source code for all pre-processing of the IoT data, including the anonymization of query parameters, can be found in Appendix A.1.

PhoneLabs

Unlike the IoT query log, the PhoneLabs logs are already anonymized. However, we still needed to filter for database query statements and exclude all non-`SELECT` queries. Similarly, we also exclude queries for the tables used internally by SQLite, those prefixed by 'sqlite' in this case. The frequency distribution of query types can also be seen in Table 5.1.

The schema of the database was not provided with the publication of the data set. However, the system would periodically emit statements that indicate the schema of one particular table. The logs were scraped to collect all these schema entries and filtered for uniqueness. Therefore, it is possible that there were more tables for which no schema-defining messages were found in the log, causing some queries to be invalid during execution. The resulting schema was exported in CSV format. The source code for all pre-processing of the PhoneLabs data can be found in Appendix A.2.

Data Set	#Tables	#Queries	#SELECT	#INSERT	#UPDATE	#DELETE	#REPLACE
IoT	20	169689	127958	2933	19346	19741	209
PhoneLabs	808	705668	557776	85439	31064	9185	0

Table 5.1: Raw data sets

5.2 SQL Query Encoding

5.2.1 Framework

The encoding scheme as described in Chapter 3 was implemented in Apache Spark using the Scala API, as it provides an excellent interface for working with query execution plans (QEPs) and supports various SQL dialects. Apache Spark implements a query execution pipeline which is depicted in Figure 5.1. This pipeline generates executable QEPs from a given SQL query. First, we load the schema corresponding to either one of the IoT and PhoneLabs data sets. Subsequently, every SQL query in the log is processed and evaluated on the database to obtain a logical plan. The logical plan is optimized by the logical optimizer to obtain multiple logical plans of which the one with the

[1]No sensitive data ever left the company server, and the collection and anonymization of the data was approved and overseen by the company's CEO.

lowest estimated execution cost is considered as the optimized logical plan. This optimization process includes, among many other steps, pushing projections down the QEP tree to reduce the cardinality of join operations. Finally, a planner takes this optimized plan and produces a physical plan, describing how to execute the plan on the cluster, which is then also optimized by a physical optimizer.

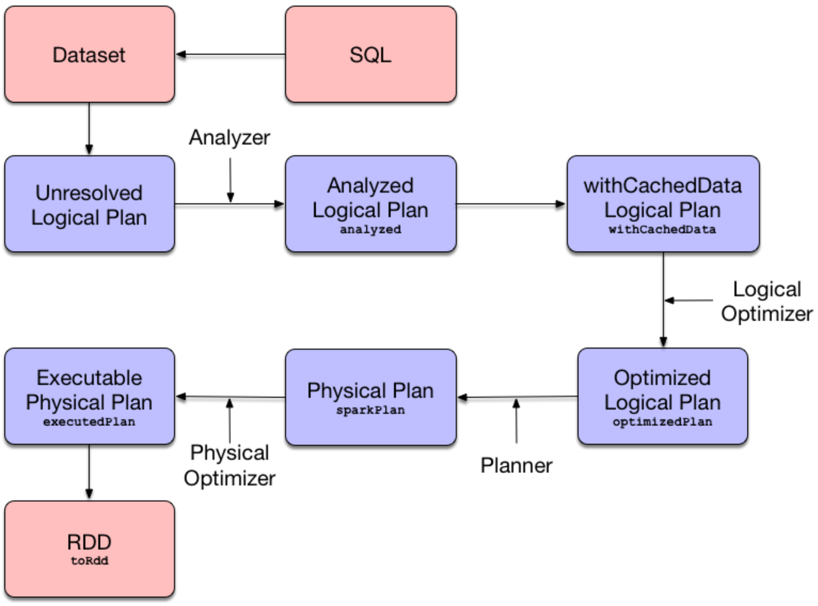


Figure 5.1: Query Execution Pipeline of Apache Spark [43]

For our purpose, the logical plan is used, which corresponds to the optimized QEP of the SQL query. The reason for this is that it allows our results to generalize to databases other than Apache Spark, such as regular centralized relational databases. However, using physical plans could potentially enable more optimizations in the case of Spark, as features would include more specific details of how queries should be executed, such as how data should be retrieved and what algorithms should be used. Obtaining multiple QEPs for a query was not yet supported, so only the most optimal logical plan was used. This logical plan would sometimes contain sub-query aliasing nodes, resulting in different partial QEPs with the same result. These were removed from the QEPs using an additional call to the built-in `EliminateSubqueryAliases` function, which is also called internally at the beginning of the optimization phase of the optimizer.

Since some of the queries resulted in SQL errors when generating the QEP, not all queries could be encoded. Upon inspection, this was usually due to incorrect aliasing, SQL dialect-specific clauses not supported by Spark, or corrupt queries. For the IoT data set, approximately 99 percent of the queries could be encoded. For the PhoneLabs data set, this was a little under 92 percent. The specific numbers are presented in Table 5.2.

Data Set	#Queries	#Processed Queries	% Encoded
IoT	111087	109619	98.67
PhoneLabs	553751	507809	91.70

Table 5.2: Encoded data sets

5.2.2 Frequency Analysis

Frequency analysis of the encoded data sets reveals some interesting details which are presented in Table 5.3 and Figures 5.2, 5.3, 5.4, and 5.5. Firstly, the number of unique partial QEPs in the IoT and PhoneLabs dataset are 5801 and 4729 respectively. Surprisingly, this number is much lower than our $n!$ estimate from before and confirms the assumption made in Section 3.1.2, which assumed how the use of query templates would reduce the number of unique partial QEPs significantly. Based on this, it was deemed unnecessary to limit the dictionary size to the top-k frequent partial QEPs and all encountered partial QEPs were encoded.

Data Set	#Unique Partial QEPs	#One-Time Partial QEPs	Avg. Set Size	Max. Set Size
IoT	5801	5634	3.78	21
PhoneLabs	4729	3548	3.02	19

Table 5.3: Partial QEP frequency analysis of the encoded data sets

Furthermore, most of the unique partial QEPs were encountered only once. As can be seen in Figures 5.2 and 5.3, the frequency distribution of partial QEPs is extremely skewed, with 5634 out of 5801 unique partial QEPs occurring only once for the IoT data set, and 3548 out of 4729 for the PhoneLabs data set. Label imbalance is a well-known problem in multi-label classification and could cause sub-optimal performance [44]. Therefore, it was decided to perform experiments with a *weighted* loss function to balance the contributions of partial QEPs with different frequencies [38].

Moreover, the number of elements in each set varies, i.e the number of operators in each QEP. Encoding complex queries with bigger QEPs results in higher cardinality sets compared to simpler queries. For the IoT dataset, approximately 19% of the queries includes one join, and 3% of the queries performs two or more joins. In contrast, only 2% of the queries in the PhoneLabs perform one join, with less than 1% including two or more joins. Hence, to determine the number of elements that should be predicted at each subsequent set, the set size distributions of the encoded queries were plotted in Figures 5.4 and 5.5. The average set size was found to be 3.78 for the IoT data set and 3.02 for the PhoneLabs data set. Consequently, we mainly focus on predicting 5 elements per subsequent set in our experiments. Nevertheless, the maximum encountered set size was 21, with many sets having more than 5 elements, especially in the IoT data set. This implies that when predicting only 5 elements per set, the model could never predict many of the encoded queries correctly. Hence, we also run experiments with more predicted elements per set: 10, 15, and 20 predicted elements.

5.2.3 Cross-Validation

After encoding the query logs, we obtain a sequence of 109619 and 507809 encoded queries for the IoT and PhoneLabs data set respectively. Ideally, we would use all the available data to train our model, but this results in extremely long running times that are unpractical for experiments. Additionally, sequences of this magnitude could face the problem of vanishing and exploding gradients as back-propagation through time becomes more difficult the longer sequences get [45]. Recall that encoder-decoder models compress the entire input sequence into a single code, so the longer the sequences are, the more difficult they become to train. Hence, sub-sequences of queries were sampled from part of the full sequence to address this problem and obtain sequences of usable lengths, while still generalizing to the entire query log and data outside the training set. In particular, we randomly sampled query sequences, i.e. windows, of length $l + m$ from the sequence of encoded queries. Here, l denotes the input sequence length and m denotes the output sequence length of the model. Each sampled window was transformed into an input-output pair $\mathbf{x}$ and $\mathbf{y}$ to train the model in an 'offline' manner. Consequently, after seeing l queries, the model can be used to make predictions about the next m queries.

Experiments were run with input sequence lengths l of 20, 30, and 40, as is typical in sequence to sequence models [14], [37], [38], [46]. In addition, we expect that these sequence lengths are likely to account for most multi-query tasks performed in a DBMS. In other words, when we predict the partial QEPs of future queries, we expect the partial QEPs from the past 20/30/40 queries to have

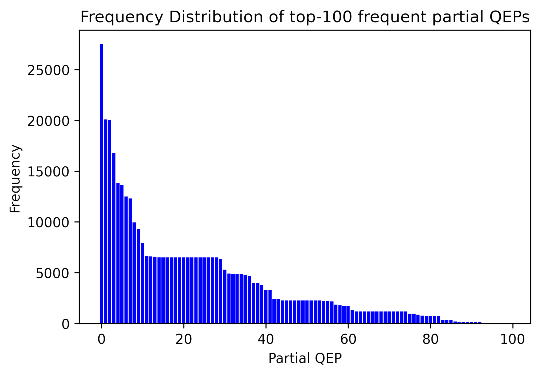


Figure 5.2: Distribution of top-100 most frequent partial QEPs in the **IoT** data set

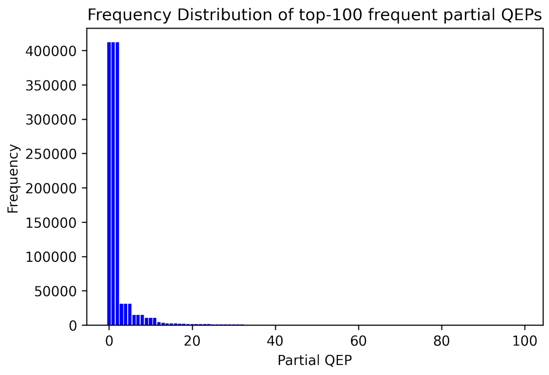


Figure 5.3: Distribution of top-100 most frequent partial QEPs in the **PhoneLabs** data set

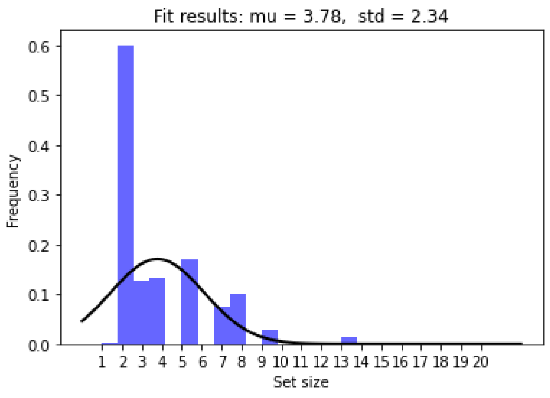


Figure 5.4: Set size distribution of the encoded queries in the **IoT** data set

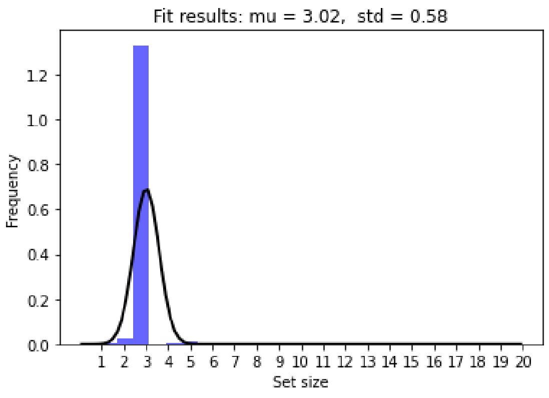


Figure 5.5: Set size distribution of the encoded queries in the **PhoneLabs** data set

a bigger effect than those from queries before it. Using longer sequences also becomes increasingly computationally intensive due to backpropagation of the loss through every recurrent unit in the sequence [45].

A known limitation of RNN sequence to sequence models is that recurrently predicting the next time steps brings errors into the input for the next time-step because there could always be errors at each predicted time-step [38]. Therefore, the more queries we consider into the future, i.e. the higher we set output sequence length m, the worse we can expect performance to be. Thus, we focus on predicting the set of partial QEPs for two queries into the future and also run experiments with slightly longer output sequence lengths of 3 and 5. Given that the predictive task revolves around predicting sequences of sets, evaluation should take into account the order of predicted sets. Therefore, evaluation metrics will compare the ground-truth and predicted partial QEPs on a per-set basis.

Since the model should be able to generalize to data outside the training data, i.e. previously unseen data, we use cross-validation, also known as out-of-sample testing. The idea of cross-validation is to reserve part of the data for evaluation of the model. In the case of sequences, it is important that we respect the temporal order of the data. Hence, we split the full sequence at a certain point in time, such that we train on the data before that point, i.e. the past, and evaluate on data after that point, i.e. the future. A disjoint training set and validation/test set were created by sampling training pairs from the first 4/5th of the sequence, and sampling validation/testing pairs from the last 1/5th of the sequence (see Figure 5.6). The 4-1 split was chosen such that we avoid unrepresentative validation/testing sets as much as possible. This way, the validation/test fold corresponds to at least one full day of query data in both data sets, rather than only the last few hours of the day for instance.

The database could experience a very different load in the last few hours of the day than during peak hours. As such, we train on 4 days of query data and evaluate on the 5th day for the PhoneLabs data set, and we train on the first 6.5 weeks of query data while evaluating on the last 1.5 weeks for the IoT data set.

Validation and testing pairs were sampled from the same split as the validation set was only used to measure performance *during* training while the test set was used to monitor the performance of the model *after* training, neither had an effect on the model during training. Experiments were run with training set sizes of 20000 and validation/test set sizes of 5000. More samples could not show enough performance increase while significantly increasing the training and evaluation time.

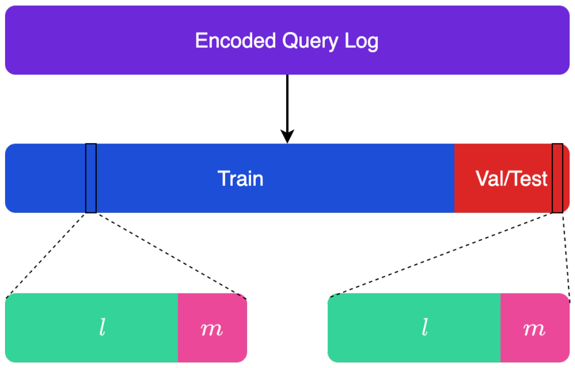


Figure 5.6: Creating training, validation and testing splits from the encoded query log

5.3 Model Implementation

The model was implemented as a Jupyter notebook and uses PyTorch, an open-source machine learning library for Python, based on code provided by Hu and He [38]. For our model, we use Gated Recurrent Unit (GRU) cells and define their parameters *input size*, *hidden size*, and *the number of layers*. In specific, the input size is set to the length of the input sequence l, the hidden size is set to 32 and denotes the number of features in the hidden state h, and the number of layers is set to 1 which defines the number of recurrent layers[1]. Since our dictionary size N is roughly 5000 for both datasets, we use 32-dimensional embeddings.

Loss Function

To address the label imbalance we use a Weighted Mean Square Error (WMSE) based on the frequency of partial QEPs in the training data. First, let n be the number of training pairs in the training set, i.e. pairs of input sequences of length l and output sequences of length m. Then, let P denote the set of distinct encoded partial QEPs, note that $|P| = N$. The WMSE is then first defined for a ground truth set $\mathbf{v}_i$ and the output of the decoder at the i-th set $o(\hat{\mathbf{v}}_i)$ as

$$WMSE(\mathbf{v}_i, o(\hat{\mathbf{v}}_i)) = \sum_k w(k)(c_k - o(\hat{c}_k))^2,$$

where c_k is the k-th entry of $\mathbf{v}_i$, and $o(\hat{c}_k)$ is the k-th entry of $o(\hat{\mathbf{v}}_i)$. Additionally, the weight w is calculated by

$$w(k) = \frac{\max_{p \in P} freq(p)}{freq(k)},$$

[1]Choosing 2 layers would mean stacking two GRUs to form a stacked GRU, such that the second GRU takes as input the outputs of the first GRU

where $freq(k)$ denotes the frequency of partial QEP k in the training data. So, less frequent partial QEPs contribute more to the loss than more frequent ones. Subsequently, we take the sum over every set in the sequence m to calculate the loss

$$\ell(\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_m\}, \{o(\hat{\mathbf{v}}_1), o(\hat{\mathbf{v}}_2), \ldots, o(\hat{\mathbf{v}}_m)\}) = \sum_{i=1}^{m} WMSE(\mathbf{v}_i, o(\hat{\mathbf{v}}_i)).$$

Consequently, we can define the empirical risk minimizer as Equation 5.1 below.

$$\hat{\theta} = \arg\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} \sum_{i=1}^{m} \sum_{k} w(k)(c_k - o(\hat{c}_k))^2. \tag{5.1}$$

Experimentation was also performed with the Mean Squared Error (MSE), Binary Cross-Entropy (BCE), and Negative Log-Likelihood (NLL) loss functions but they could not show better performance than the WMSE.

Optimization

For optimization of the model, the Adam optimizer for stochastic optimization [47] was used, as it performed better than SGD, Adadelta, and RMSprop optimization. Hyperparameters were tuned with grid search, where the learning rate of 0.001 resulted in the best performance and fastest convergence in combination with beta coefficients set to 0.9, 0.99 and epsilon set to 10^{-11}. The model was trained for 10 epochs and weights were updated after every input-output training pair.

Regularization

Some experimentation with weight decay (L2 regularization) and dropout (5 %, 10%, and 20%) was performed but could not show better performance.

Use of the Dependency Graph

As was mentioned before, the dependency graph could be used in two pre-defined ways. First, the dependency graph could be used at each recurrent unit to include the implicit elements of the top-k predictions in the output. This output is then recurrently fed as input into the next recurrent unit and repeated for every encoded query in the sequence. Consequently, the dependency graph will affect the training and optimization of the model. This is perhaps reminiscent of the concept of teacher forcing, where during training the ground truth is used as input of a time-step, rather than the model output from a prior time step. However, instead of using the ground truth elements, we use the ground truth *relation* between elements to add partial QEPs to the predicted set. As such, incorrect predictions also introduce more errors in the input of the next unit. In contrast to teacher forcing, there is no obvious discrepancy between training and inference as the ground truth relation between elements remains known and unchanged during inference.

Alternatively, the dependency graph can be used only as a post-processing step to improve the completeness of predictions. In this case, the model's architecture is unaltered, and it is and trained independently of the dependency graph. During inference, the top-k predictions are used to consult the dependency graph and include the missing elements in the final prediction. Experiments were run to test the effectiveness of both approaches.

5.4 Model Evaluation

5.4.1 Evaluation Methods

To evaluate the results of our predictive model we compare its performance with three other methods. These methods are outlined below:

- **RandomSet** predicts k random distinct partial QEPs for each subsequent set by randomly selecting k distinct indices from $\{1, 2, \ldots, N\}$. In other words, it randomly guesses the set of partial QEPs at each subsequent query.
- **TopKFrequent** predicts the k most frequent partial QEPs for all subsequent sets.
- **ExpectRepeat** predicts the most recently seen set of partial QEPs for all subsequent sets, irrespective of k.

5.4.2 Evaluation Metrics

We also use several measurements to evaluate the performance of our predictive model. The prediction for each subsequent set can be considered as multi-class classification, for which there are some commonly used metrics. Perhaps the most common ones are Recall and Precision [48], [49], which are calculated from the confusion matrix. The confusion matrix summarizes the performance of the model by comparing the predicted class with the real class for each instance of the data. A labeled confusion matrix is presented in Table 5.4. In our case, we can define the true positives as the partial QEPs in the predicted set which also occur in the ground truth set. Similarly, the true negatives include the partial QEPs which were not predicted and don't occur in the ground truth set. False positives denote the set of partial QEPs which were wrongly predicted, meaning that they were not present in the set of partial QEPs corresponding to the future query. False negatives describe the partial QEPs of the future query which we failed to predict.

	Predicted Class	
Real Class	**True Positive**	False Negative
	False Positive	True Negative

Table 5.4: Confusion Matrix

Recall & Precision

Recall is the fraction of true positives over all predictions. In other words, the percentage of total relevant results correctly classified. As such, recall can be interpreted as the fraction of partial QEPs in the next query that was correctly predicted.

$$\text{Recall} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}}$$

Precision is the fraction of true positives over all positives, i.e. the percentage of results that are relevant. Therefore, precision can be interpreted as the fraction of predicted partial QEPs occurring in the next query.

$$\text{Precision} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}}$$

Generally, there is a trade-off between precision and recall. One can optimize a model for recall by making more predictions, and thus increasing the fraction of true positives over all predictions, at the cost of more false positives and lower precision. To this end, we compare recall and precision for the same number of predictions.

F-score

F-score is another metric for measuring the model's performance. It is calculated from the recall and precision, and there exists both a balanced variant and a weighted variant. The balanced variant, also called the F_1-score, is the harmonic mean of the recall and precision, such that both are valued equally in the calculation. It is calculated from the recall and precision as

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}.$$

When presented together with the recall and precision, it conveys little additional information, as higher values result in a higher F_1-score and lower values in a lower F_1 score. However, the weighted variant allows us to place more importance on either recall or precision through a parameter β. This F_β-score is calculated as

$$F_\beta = (1 + \beta^2) \cdot \frac{\text{precision} \cdot \text{recall}}{(\beta^2 \cdot \text{precision}) + \text{recall}},$$

where β denotes how many times recall should be considered more important as precision. Note that when $\beta = 1$, it is equivalent to the F_1-score.

Normalized Discounted Cumulative Gain (NDCG)

The Normalized Discounted Cumulative Gain (NDCG) is a metric for ranking the quality of a prediction while taking the order of predictions into account [50]. It is often used to assess the relevance of search engine results. While recall and precision only consider binary relevance, i.e. a partial QEP is correctly predicted or it is not, NDCG allows for varying degrees of relevancy and incorporates a discount function over the rank. It allows us to measure the ability of our inference algorithm, Algorithm 5, to predict relevant partial QEPs before irrelevant ones. The NDCG is calculated using the discounted cumulative gain (DCG) and the ideal discounted cumulative gain (IDCG). These are specified by

$$DCG_p = \sum_{i=1}^{p} \frac{rel_i}{\log_2(i+1)},$$

and

$$IDCG_p = \sum_{i=1}^{|REL_p|} \frac{2^{rel_i} - 1}{\log_2(i+1)},$$

where p denotes the position in the list and REL_p is a list of all relevant predictions in the dictionary up to position p. Subsequently, the NDCG is computed by

$$NDCG_p = \frac{DCG_p}{IDCG_p},$$

to obtain a normalized value between 0 and 1. We calculate the NDCG with a relevance score of 1 if the predicted item is present in the ground truth and 0 if not. Thus, the more correctly predicted partial QEPs are higher in the top-k predicted elements of our inference algorithm, the higher the NDCG score.

Hit Rate (HR)

Lastly, we use a variant of Hit Rate (HR) to measure the performance. HR represents the ratio of testing instances for which we are able to correctly predict at least one element in each subsequent set [51]. HR is defined as

$$HR = \frac{\#hits}{n},$$

where a hit is counted if at least one element is correctly predicted in all subsequent sets of the output sequence and n denotes the number of testing samples. Unlike recall, HR conveys information on how often the model is able to make at least some partially correct predictions for future queries.

5.5 A Practical Setting

While the previously described evaluation methods and metrics of the model could be used as an indication of the models' usefulness, it doesn't fully convey how the model would perform in a practice. In particular, recall and precision make no distinction other than the correctness of predicted partial QEPs. A high recall would indicate that most of the relevant partial QEPs were predicted. However, what if most predictions correspond to base table operators? It would suggest to simply keep some particular tables cached, despite the possibility of them being too large to fit in memory. Consider also partial QEPs that are costly to compute. If these are rarely predicted, deploying the model in practice would only result in a marginal increase in query performance. Similarly, a lower precision means that more irrelevant partial QEPs are predicted. Caching irrelevant partial QEPs comes at the cost of occupying memory space and could prevent relevant partial QEPs from being cached at the time of query arrival. Therefore, some experiments were performed to determine the *usefulness* of the model in a practical setting.

Even though there are many ways that a DBMS could exploit the model's predictions, we will focus on the case of data caching. As such, we investigate the query answering time when the model is used to make an 'intelligent' decision on what data should remain in memory or pre-computed. Subsequently, the cached data can be used as a surrogate for certain query operations such that queries can be answered more rapidly. In our experiments, the testing data set is used as a simulated query workload for which we compare the effectiveness of several optimization methods.

5.5.1 Cost Estimation

Since every partial QEP corresponds to some intermediate result of the query, every partial QEP has an associated cost: the computation time and size in bytes of its intermediate result. However, recall that queries were transformed into query templates before being encoded into a set of partial QEPs. As such, partial QEPs don't include the filter parameters, e.g. a query `SELECT * FROM cities WHERE name='Eindhoven'` was transformed into `SELECT * FROM cities WHERE name='?'`, and this is reflected in the partial QEPs of a query. Therefore, the computation time and byte sizes of a partial QEP don't correspond to that of the query exactly. Nevertheless, for larger tables, the *type* of operator would have a bigger impact on the cost than the filter parameters. For instance, join operations have a known time complexity of $O(n^2)$, and will require more computation time for larger values of n than filter operations that have a time complexity of $O(n)$, where n is the number of rows in the table. Naturally, the computation time is also dependent on the host machine, faster computers will achieve faster computation times. Hence, to obtain estimates of the cost values of the query and its partial QEPs, we need to simulate the query by replacing the query filter placeholders with some value.

The cost values were estimated by creating a dummy database with the corresponding schema, and subsequently entering 100,000 dummy rows into each table[1]. The dummy rows also follow the schema definition to create a database that closely represents the real database. A form of randomness was introduced in the row attributes to create distinct rows, while also ensuring that there would be some shared keys between tables, so that join operations don't match either zero or all rows. Thereby, rows were generated with values randomly selected between 1 and 100,000, cast into their respective

[1]More than 100,000 rows resulted in memory problems on the testing machine

types, while ensuring that at least 10 rows would include only the 1 or '1' values. Subsequently, the placeholders in the query templates were replaced with a numeric 1 or string '1' depending on the data type. For example, the query template `SELECT * FROM cities WHERE name='?'` would become `SELECT * FROM cities WHERE name='1'`, and it would match at least 10 rows in the 'cities' table. Due to the extreme number of tables in the PhoneLabs database, it was deemed unpractical for this experiment and only the IoT data set was considered.

Subsequently, every unique partial QEP in the encoded query log of the IoT data set was executed on the dummy database to measure the computation time in milliseconds and size in bytes of the resulting table. Consequently, we were able to associate every partial QEP with a computation time and size of its result in bytes. For these experiments, we used a desktop PC with a 3.5 GHz Quad-Core 64-bit CPU architecture, 16 GB of RAM, and with the database stored on an SSD that supports 94.000 I/O operations per second.

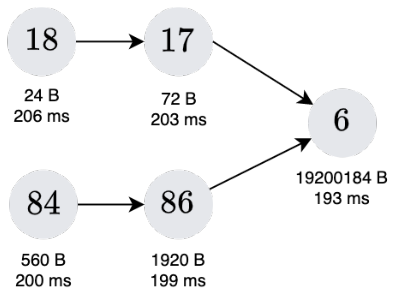


Figure 5.7: Example dependency graph where nodes have an associated cost

The associated cost values were added to each node in the dependency graph, an example of this is given in Figure 5.7. As can be seen in Figure 5.7, two queries are encoded to $\mathcal{S}_1 = \{6, 84, 86\}$ and $\mathcal{S}_2 = \{6, 17, 18\}$ and every node has an associated cost. Recall that edges of the graph imply the presence of partial QEPs. For example, the partial QEP with index 84 implies the presence of the partial QEP with index 86. Furthermore, the query template encoded to $\mathcal{S}_1$ requires 200 ms to compute and its result is 560 bytes. This corresponds to executing the partial QEP that is not implied by other partial QEPs in $\mathcal{S}_1$, the one with index 84. The difference between the partial QEP with index 84 and 86 is only 1 ms, while the difference between 86 and 6 is 6 ms. From this, we can deduce that the partial QEP corresponding to index 6 contributes the most to the query execution time. In fact, it corresponds to a table scan which takes considerable time due to reading from disk. Consequently, if the table corresponding to partial QEP with index 6 was cached in memory, it would have a large impact on both queries' answering time. However, possibly more optimization could be achieved if the results partial QEPs 18 and 84 were in memory, reducing the computation time to zero, excluding the additional time needed to infer the model and account for the filter parameters in the original queries.

One can also observe a relation between the computation time of connected nodes. Nodes without incoming edge represent the full QEP and have the highest computation time, and every node in the directed chain has a computation time less than or equal to that of the previous node. The reverse is not the case for byte size, as join operations might increase the number of rows.

5.5.2 Optimization Methods

The experimental performance of the model was assessed by implementing several optimization methods, including two commonly used caching strategies. Namely, the Least-Recently-Used (LRU) and First-in-First-Out (FIFO) replacement policies, which are also used in database environments [52]. Additionally, we use the model predictions in an attempt to improve upon these two caching policies by selectively keeping items cached. Lastly, we use the model in a setting where we assume that the predicted partial QEPs can be pre-computed before the arrival of the query. All the considered methods are summarized in the following list:

- **None**, serves as baseline
- **Least-Recently-Used (LRU)**, a cache that evicts the least recently used partial QEPs from the cache
- **First-In-First-Out (FIFO)**, a cache that evicts the partial QEPs in order of arrival from the cache
- **Least-Recently-Used (LRU) with model**, a cache that evicts the least recently used partial QEPs from the cache that are not predicted for the next *m* queries
- **First-In-First-Out (FIFO) with model**, a cache that evicts the partial QEPs in order of arrival from the cache that are not predicted for the next *m* queries
- **Model**, pre-compute the predicted partial QEPs of the next *m* queries and cache the results

5.5.3 Simulation

The methods in Section 5.5.2 were implemented in a simulation framework written in Python, which can be found in Appendix A.5. This framework considers the testing data set as an input sequence of queries (21898 queries for the IoT data set) and uses either one of the methods in an attempt to optimize the process by reducing the computation time. The cache capacity is fixed and given as a parameter of the simulation. The memory capacity of the cache is an important parameter, as setting it too high would allow any method to keep everything cached in memory. It was observed that tables with 100,000 dummy rows were approximately 25 MB, which is trivial to cache with today's available memory sizes. However, database tables can contain millions or even billions of rows of data, which are less realistic to keep in memory. Furthermore, the sum over every unique partial QEP's size in bytes amounted to approximately 2 GB. As such, a cache capacity of 2GB would already enable every encountered partial QEP to be cached. Therefore, to create a more realistic scenario, experiments were run with cache sizes of 25, 50, 75, and 100 MB to allow the caching of approximately 1 through 4 tables respectively.

For the simulation, every encoded query in the testing data set is considered sequentially. At every query arrival, partial QEPs were added to the cache and evicted from the cache based on the selected strategy, the associated byte size of the partial QEP, and the memory capacity of the cache. Only partial QEPs with sizes less or equal to the cache capacity were be added to the cache. The optimization of query answering was simulated by checking if any partial QEP in the query was found in cache, and subtracting the computation time from that of the full query. Since multiple partial QEPs of a query could be in cache, only the highest level partial QEPs, i.e. the partial QEPs in the query not implied by other partial QEPs in the cache, were subtracted from the computation time of the query. In other words, we use the cached partial QEPs that give the biggest performance increase. Implicatively, we assume the cost of accessing the cache to be zero. This assumption is sensible as accessing memory is approximately 100,000 times faster than accessing data on disk [12]. Therefore, accessing the cache has negligible cost compared to computing the result of a partial QEP and performing the necessary I/O operations.

Chapter 6

Results & Discussion

Next, the most important results from experiments described in the previous chapter are presented and discussed. First, we will address the model performance and compare it with the prediction methods described in Section 5.4.1. Afterward, the effect of the dependency graph, repeated element component, sequence lengths, and partial query execution plan (QEP) size will be discussed. Lastly, the results of placing the model in a practical setting will be addressed. For each experiment, the model checkpoint at the epoch with the best performance was used for evaluation. In addition, all experiments were performed on the testing set, with data never seen before by the model. Detailed tables for all the experiments can be found in Appendix B.

6.1 Model Performance

The results of the model's best performance on the Internet of Things (IoT) and PhoneLabs data sets are presented in Tables 6.1 through 6.4. Several observations can be made from the results. First of all, for the IoT data set, the model achieves significantly higher recall than other methods. As can be seen in Table 6.1, when we predict 5 partial QEPs per set, and then use the dependency graph to include the implicit partial QEPs, we find that we are able to correctly predict over 82% of the partial QEPs of the future queries. Precision, normalized discounted cumulative gain (NDCG), and hit rate (HR) are also significantly better than other methods for $k = 5$. As expected, precision decreases and recall increases when the value of k is increased. However, recall improves only slightly to 87% while precision drops to as low as 13% for $k = 20$. For values of $k = 10$ or higher, TopKFrequent and ExpectRepeat outperform the model in terms of precision and HR. The HR of the TopKFrequent method reveals that in roughly 92% of the testing instances, every subsequent set has at least one element from the top-k most frequent partial QEPs. As for NDCG, it can be seen that the model outperforms other methods for all values of k, indicating that it is better at predicting relevant items before irrelevant ones. The model, TopKFrequent and ExpectRepeat all outperform RandomSet significantly in every metric. In summary, the model outperforms other methods significantly for all values of k when the goal is to predict as many of the ground truth partial QEPs as possible.

For the PhoneLabs data set, a much smaller difference between the prediction methods can be observed. Looking at Table 6.2, it may seem that the model achieves really good performance, but ExpectRepeat and TopKFrequent achieve almost the same performance. In addition, ExpectRepeat significantly outperforms the model in terms of precision, because it predicts the same number of elements as were observed in the previous query irrespective of k. These results indicate that the PhoneLabs data set is subject to an extremely high amount of repeated encoded queries, given that predicting a repeated encoded query is correct approximately 93% of the time. In other words, the PhoneLabs data set is not just subject to simple queries, but also subject to simple query patterns that are easily exploited by simple methods. Still, the model achieves between 2 - 4% higher recall, HR, and NDCG compared to other methods, so if maximizing these metrics is a priority, the model is still the best prediction method. However, one can ask whether or not it is worth the overhead of using the model compared to the other, relatively simple prediction methods in this case.

The F_β-score for different values of β was also examined, i.e. how does the model perform when recall and precision have different levels of importance. The β parameter denotes how many times recall should be considered more important than precision. Results for $\beta \in \{0.5, 1.0, 1.5, 2.0\}$ and $k \in \{5, 10, 15, 20\}$ are summarized in Tables 6.3 and 6.4. As can be seen in Table 6.3, the model outperforms all other methods on the IoT data set for all values of β when $k = 5$. For higher values of k, a higher value of β is needed to outperform the other methods. Since increasing the value of k positively affects recall, and negatively affects precision, this was expected. For the PhoneLabs data set, the model outperforms the RandomSet and TopKFrequent methods for all values of β and k, except for $k = 20$. However, the ExpectRepeat method achieves significantly higher precision, so for all values of β it is better than the model. Not only do these results show that the PhoneLabs data set is subject to a very high amount of repeated query templates, but also that precision is difficult for the model to achieve. On the other hand, results show that the rather primitive method of ExpectRepeat in combination with our encoding scheme could be very powerful in some databases.

The recall and precision trade-off still remains to be discussed. One could argue that recall is more important than precision. As the goal is to increase query performance, predicting as many relevant partial QEPs as possible implies that a database management system (DBMS) could optimize for more queries, and to a greater extent. As was also demonstrated in Tables 6.1 and 6.2, the model achieves recall scores of over 80% for all values of k in both data sets. In other words, the model predicts over 80% of the partial QEPs correctly for each subsequent query on average. Thereby, many details of future queries are revealed to the DBMS before their arrival. However, these details are accompanied by irrelevant information, that cannot be distinguished from relevant information until the arrival of future queries. This is why precision is also important, which conveys the fraction of relevant details that are predicted. For both data sets, this number is generally lower than 50%, reaching as low as 13% when $k = 20$. The problem with recall and precision metrics, as well as F_β-score, is that it considers every prediction to be equally valuable; a partial QEP is correctly predicted or it is not. In reality, every predicted element corresponds to a partial QEP, which has an associated size in bytes and computation time of its result. Therefore, whether a DBMS could optimize for every prediction, with the possibility of up to 87% being futile, depends on the available memory and computation capacity. With enough memory and computation capacity, a DBMS could prepare for future queries at the cost of many useless memory allocations and pre-emptive calculations. But, some partial QEPs may correspond to extremely large relations, which could realistically not fit into memory at all. In summary, it is difficult to determine the actual *usefulness* of the model based on only these metrics. Experiments of the model in a practical setting will study the extent to which the recall-precision trade-off affects the model's usefulness.

IoT	$N = 5801, l = 20, m = 2$							
	$k = 5$				$k = 10$			
Method	**Recall**	**Precision**	**NDCG**	**HR**	**Recall**	**Precision**	**NDCG**	**HR**
RandomSet	0.0009	0.0007	0.0015	0.0000	0.0023	0.0008	0.0033	0.0000
TopKFrequent	0.3049	0.1526	0.2606	0.2200	0.5132	0.1396	0.3463	**0.8438**
ExpectRepeat	0.3539	0.3591	0.3762	0.3368	0.3539	**0.3591**	0.3695	0.3368
Model	**0.8216**	**0.4323**	**0.6997**	**0.7556**	**0.8286**,	0.2474	**0.6671**	0.7720
	$k = 15$				$k = 20$			
	Recall	**Precision**	**NDCG**	**HR**	**Recall**	**Precision**	**NDCG**	**HR**
RandomSet	0.0026	0.0006	0.0037	0.0000	0.0036	0.0007	0.0052	0.0002
TopKFrequent	0.5721	0.1122	0.3709	**0.9246**	0.6175	0.1001	0.3915	**0.9246**
ExpectRepeat	0.3539	**0.3591**	0.3687	0.3368	0.3539	**0.3591**	0.3687	0.3368
Model	**0.8521**	0.1705	**0.6345**	0.8224	**0.8727**	0.1320	**0.6300**	0.8302

Table 6.1: Evaluation of different methods on the **IoT** data set

PhoneLabs	$N = 4729, l = 20, m = 2$							
	$k = 5$				$k = 10$			
Method	**Recall**	**Precision**	**NDCG**	**HR**	**Recall**	**Precision**	**NDCG**	**HR**
RandomSet	0.0010	0.0006	0.0014	0.0000	0.0022	0.0007	0.0031	0.0000
TopKFrequent	0.8918	0.5351	0.8752	0.8984	0.9182	0.2755	0.8877	0.9164
ExpectRepeat	0.9268	**0.9271**	0.9276	0.9088	0.9268	**0.9271**	0.9276	0.9088
Model	**0.9487**	0.5604	**0.9426**	**0.9350**	**0.9560**	0.2809	**0.9454**	**0.9444**
	$k = 15$				$k = 20$			
	Recall	**Precision**	**NDCG**	**HR**	**Recall**	**Precision**	**NDCG**	**HR**
RandomSet	0.0036	0.0007	0.0050	0.0002	0.0044	0.0007	0.0062	0.0000
TopKFrequent	0.9388	0.1875	0.8953	0.9282	0.9447	0.1414	0.8972	0.9362
ExpectRepeat	0.9268	**0.9271**	0.9276	0.9088	0.9268	**0.9271**	0.9276	0.9088
Model	**0.9601**	0.1901	**0.9466**	**0.9522**	**0.9640**	0.1386	**0.9483**	**0.9580**

Table 6.2: Evaluation of different methods on the **PhoneLabs** data set

IoT	$N = 5801, l = 20, m = 2$							
	$k = 5$				$k = 10$			
Method	$\beta = 0.5$	$\beta = 1.0$	$\beta = 1.5$	$\beta = 2.0$	$\beta = 0.5$	$\beta = 1.0$	$\beta = 1.5$	$\beta = 2.0$
RandomSet	0.0007	0.0008	0.0008	0.0009	0.0009	0.0012	0.0015	0.0017
TopKFrequent	0.1695	0.2034	0.2333	0.2542	0.1634	0.2195	0.2814	0.3343
ExpectRepeat	0.3580	0.3565	0.3555	0.3549	**0.3580**	0.3565	0.3555	0.3549
Model	**0.4776**	**0.5665**	**0.6433**	**0.6962**	0.2878	**0.3810**	**0.4809**	**0.5637**
	$k = 15$				$k = 20$			
	$\beta = 0.5$	$\beta = 1.0$	$\beta = 1.5$	$\beta = 2.0$	$\beta = 0.5$	$\beta = 1.0$	$\beta = 1.5$	$\beta = 2.0$
RandomSet	0.0007	0.0010	0.0013	0.0016	0.0008	0.0012	0.0016	0.0020
TopKFrequent	0.1337	0.1876	0.2530	0.3144	0.1203	0.1723	0.2384	0.3036
ExpectRepeat	**0.3580**	**0.3565**	0.3555	0.3549	**0.3580**	**0.3565**	**0.3555**	0.3549
Model	0.2030	0.2841	**0.3821**	**0.4735**	0.1590	0.2293	0.3201	**0.4112**

Table 6.3: F_β-scores for different methods and different values of β on the **IoT** data set

PhoneLabs	$N = 4729, l = 20, m = 2$							
	$k = 5$				$k = 10$			
Method	$\beta = 0.5$	$\beta = 1.0$	$\beta = 1.5$	$\beta = 2.0$	$\beta = 0.5$	$\beta = 1.0$	$\beta = 1.5$	$\beta = 2.0$
RandomSet	0.0007	0.0008	0.0008	0.0009	0.0008	0.0011	0.0013	0.0015
TopKFrequent	0.5816	0.6689	0.7400	0.7869	0.3203	0.4238	0.5345	0.6261
ExpectRepeat	**0.9270**	**0.9269**	**0.9269**	**0.9269**	**0.9270**	**0.9269**	**0.9269**	**0.9269**
Model	0.6104	0.7046	0.7820	0.8332	0.3271	0.4342	0.5496	0.6457
	$k = 15$				$k = 20$			
	$\beta = 0.5$	$\beta = 1.0$	$\beta = 1.5$	$\beta = 2.0$	$\beta = 0.5$	$\beta = 1.0$	$\beta = 1.5$	$\beta = 2.0$
RandomSet	0.0008	0.0012	0.0016	0.002	0.0008	0.0012	0.0017	0.0021
TopKFrequent	0.2232	0.3126	0.4204	0.5212	0.1704	0.246	0.3438	0.4422
ExpectRepeat	**0.9270**	**0.9269**	**0.9269**	**0.9269**	**0.9270**	**0.9269**	**0.9269**	**0.9269**
Model	0.2264	0.3174	0.4274	0.5304	0.1672	0.2424	0.3403	0.4400

Table 6.4: F_β-scores for different methods and different values of β on the **PhoneLabs** data set

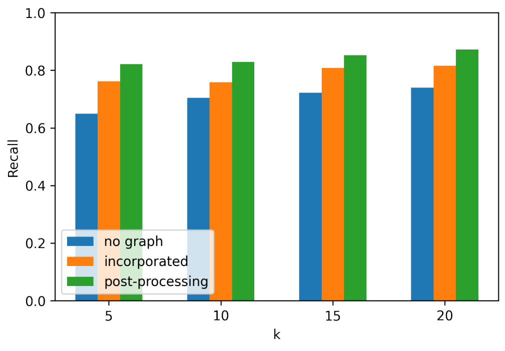


Figure 6.1: Recall scores of **IoT** data set by use of dependency graph and *k*

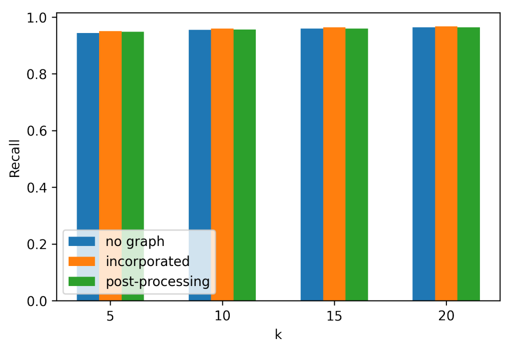


Figure 6.2: Recall scores of **PhoneLabs** data set by use of dependency graph and *k*

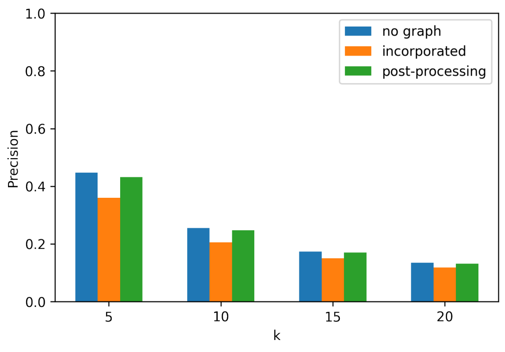


Figure 6.3: Precision scores of **IoT** data set by use of dependency graph and *k*

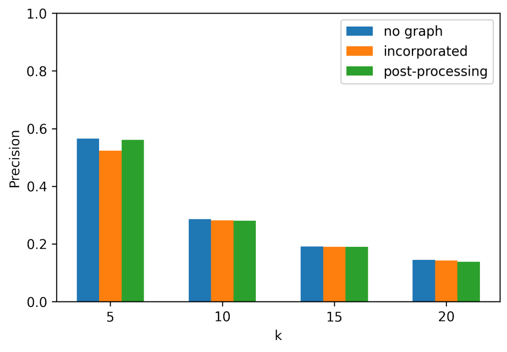


Figure 6.4: Precision scores of **PhoneLabs** data set by use of dependency graph and *k*

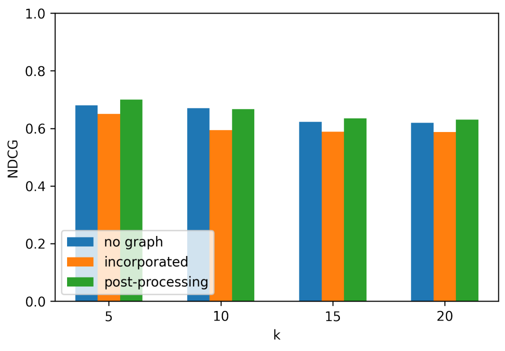


Figure 6.5: NDCG scores of **IoT** data set by use of dependency graph and *k*

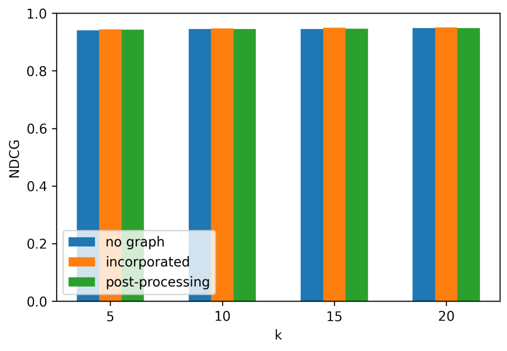


Figure 6.6: NDCG scores of **PhoneLabs** data set by use of dependency graph and *k*

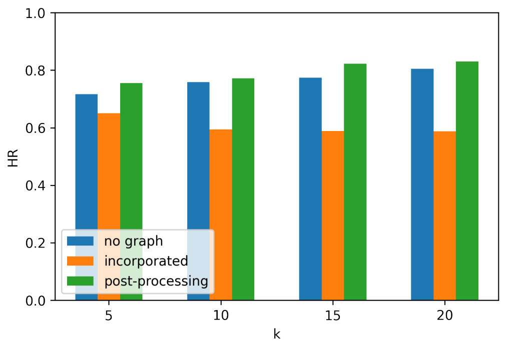


Figure 6.7: HR scores of **IoT** data set by use of dependency graph and *k*

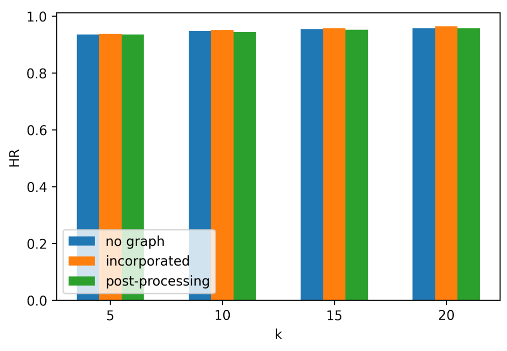


Figure 6.8: HR scores of **PhoneLabs** data set by use of dependency graph and *k*

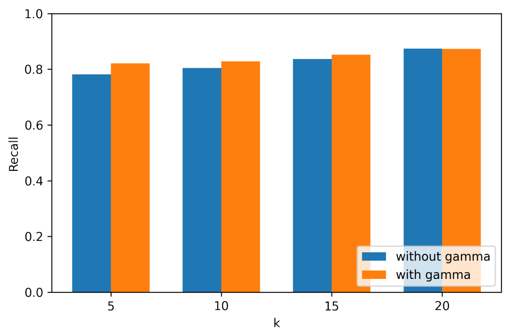


Figure 6.9: Recall scores of **IoT** data set by use of repeated element component γ and k

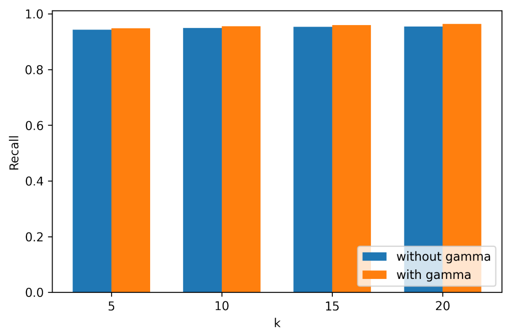


Figure 6.10: Recall scores of **PhoneLabs** data set by use of repeated element component γ and k

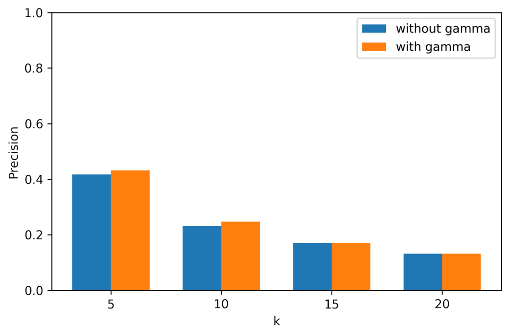


Figure 6.11: Precision scores of **IoT** data set by use of repeated element component γ and k

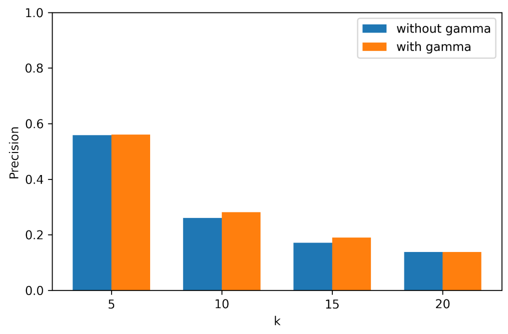


Figure 6.12: Precision scores of **PhoneLabs** data set by use of repeated element component γ and k

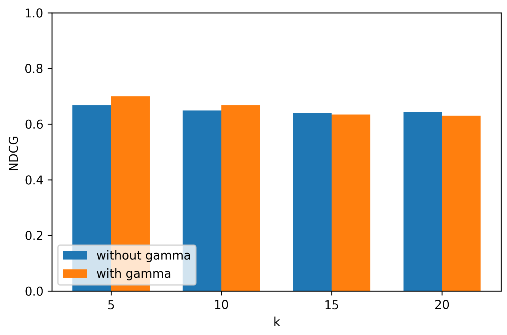


Figure 6.13: NDCG scores of **IoT** data set by use of repeated element component γ and k

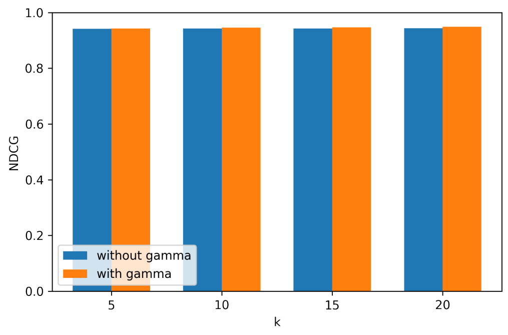


Figure 6.14: NDCG scores of **PhoneLabs** data set by use of repeated element component γ and k

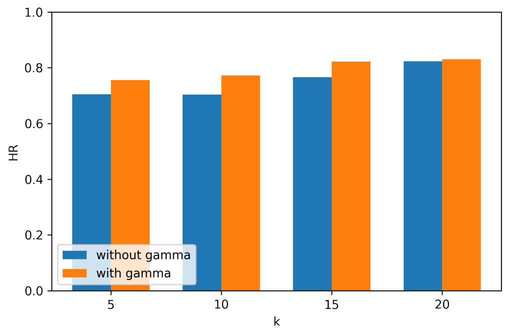


Figure 6.15: HR scores of **IoT** data set by use of repeated element component γ and k

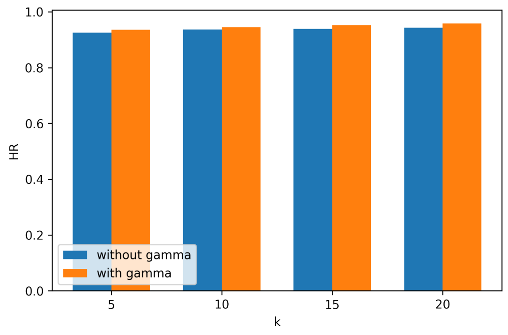


Figure 6.16: HR scores of **PhoneLabs** data set by use of repeated element component γ and k

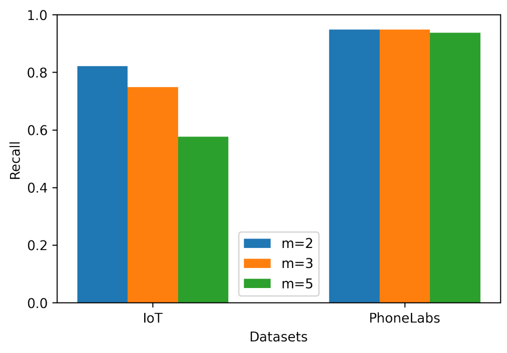


Figure 6.17: Recall scores by data set and m, with $l = 20$ and $k = 5$

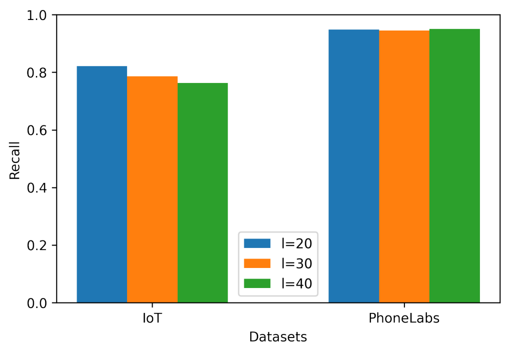


Figure 6.18: Recall scores by data set and l, with $m = 2$ and $k = 5$

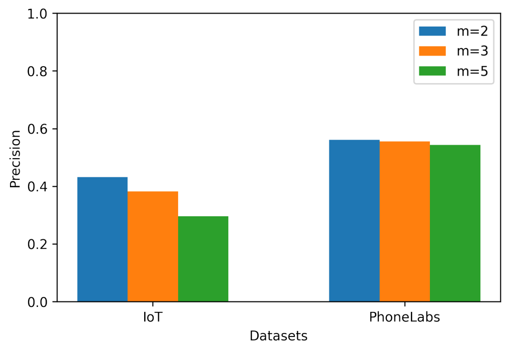


Figure 6.19: Precision scores by data set and m, with $l = 20$ and $k = 5$

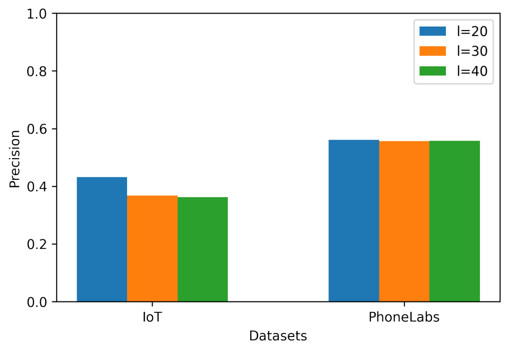


Figure 6.20: Precision scores by data set and l, with $m = 2$ and $k = 5$

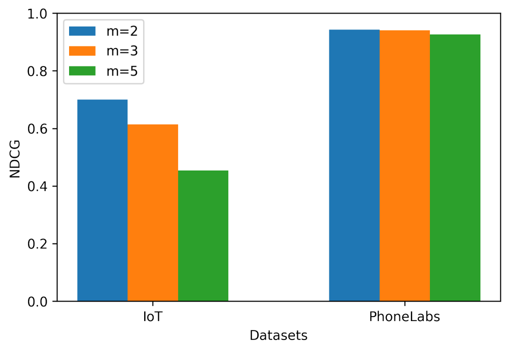


Figure 6.21: NDCG scores by data set and m, with $l = 20$ and $k = 5$

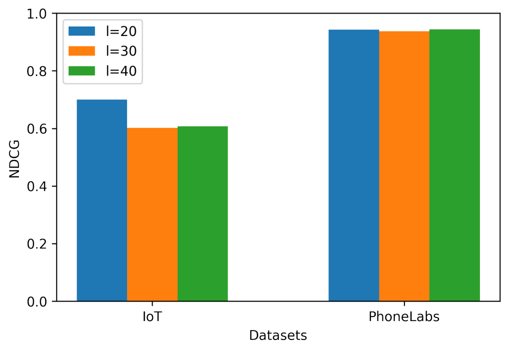


Figure 6.22: NDCG scores by data set and l, with $m = 2$ and $k = 5$

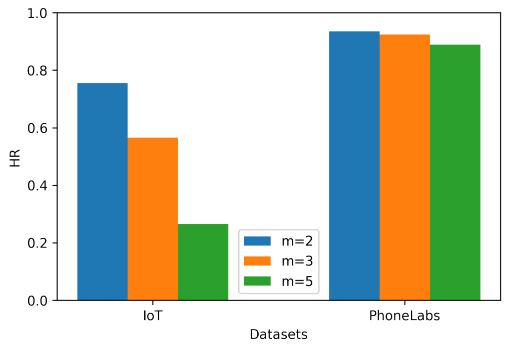


Figure 6.23: HR scores by data set and m, with $l = 20$ and $k = 5$

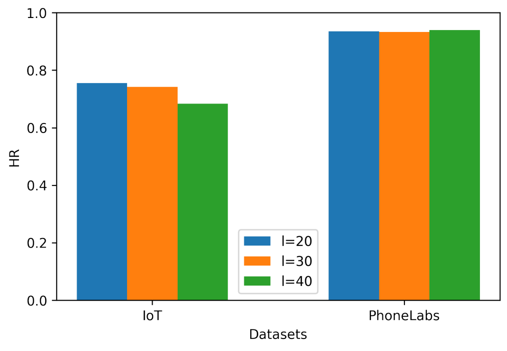


Figure 6.24: HR scores by data set and l, with $m = 2$ and $k = 5$

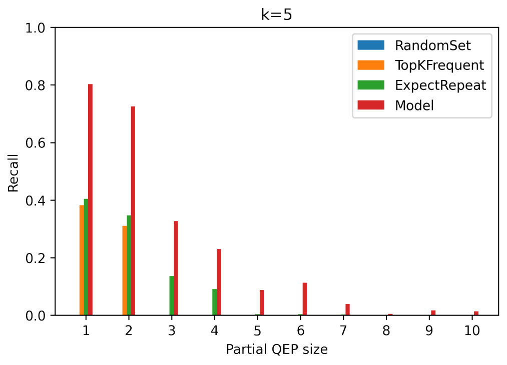


Figure 6.25: Recall scores of **IoT** data set by partial QEP size and method, with $k = 5$

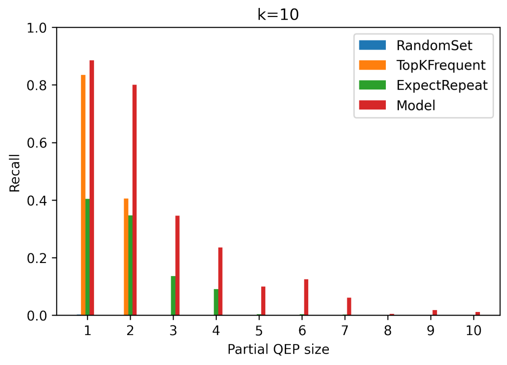


Figure 6.26: Recall scores of **IoT** data set by partial QEP size and method, with $k = 10$

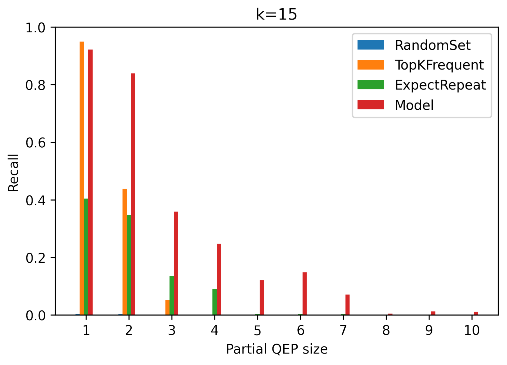


Figure 6.27: Recall scores of **IoT** data set by partial QEP size and method, with $k = 15$

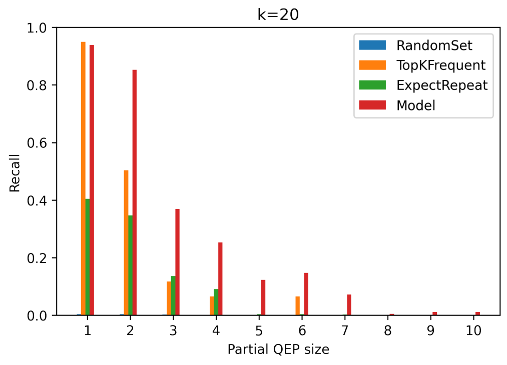


Figure 6.28: Recall scores of **IoT** data set by partial QEP size and method, with $k = 20$

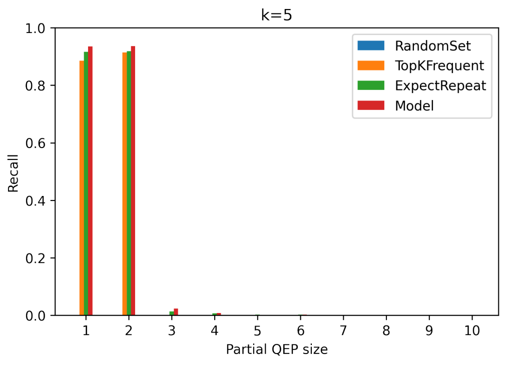


Figure 6.29: Recall scores of **PhoneLabs** data set by partial QEP size and method, with $k = 5$

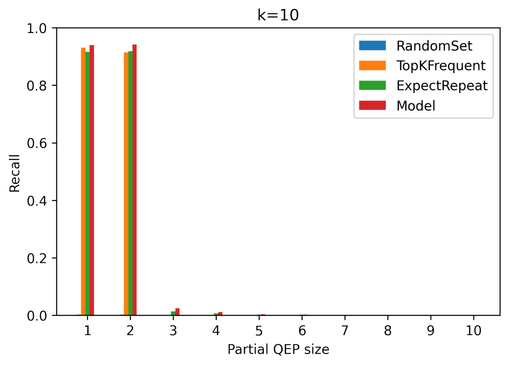


Figure 6.30: Recall scores of **PhoneLabs** data set by partial QEP size and method, with $k = 10$

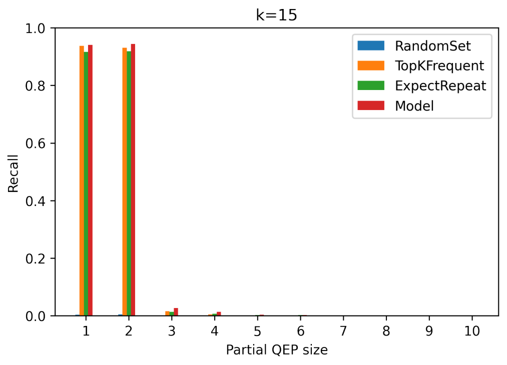


Figure 6.31: Recall scores of **PhoneLabs** data set by partial QEP size and method, with $k = 15$

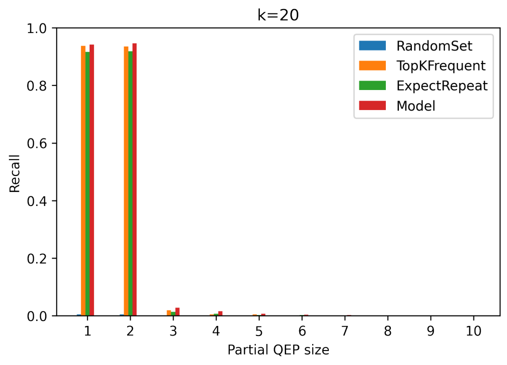


Figure 6.32: Recall scores of **PhoneLabs** data set by partial QEP size and method, with $k = 20$

6.2 Effect of the Dependency Graph

Figures 6.1-6.8 showcase the effect of the dependency graph on model performance. Experiments were performed using input sequences of length $l = 20$ and output sequence lengths of $m = 2$. Detailed results can be seen in Tables 6.1, 6.2, and Appendix B.

For the IoT data set, incorporating the dependency graph significantly improves recall compared to the unaltered model architecture. The improvements in recall can be seen for all values of k. Unfortunately, this also comes at the cost of lower precision. For $k = 5$ the difference in precision is most notable and gradually decreases for higher values of k. Since the dependency graph increases the number of elements in the predicted set by including the implicit elements for each of the k elements, the drop in precision is expected. The increasingly smaller drop in precision for larger values of k indicates that the error introduced by the dependency graph has a smaller effect for larger values of k. NDCG and HR are also significantly worse for all values of k, being 5-10% less on average.

When the dependency graph is used as a post-processing step instead of incorporating it in the model, an even bigger improvement in recall can be seen. As can be seen in Figures 6.1 and 6.3, recall increases from approximately 64% to 82% with a minimal decrease in precision for the IoT data set. A similar improvement in recall can be seen for higher values of k. However, the drop in precision is surprisingly small given the significant boost in recall performance, and much better than incorporating the dependency graph in the model directly. In addition, HR and NDCG improve when using the dependency graph during inference, indicating that the dependency graph has a positive effect on predicting relevant elements before irrelevant ones and predicting at least one element correctly in all k subsequent sets. The difference in HR also demonstrates that there are cases where the model predicts a set of partial QEPs of which none are correct, but at least one of the implicit partial QEPs is correct, resulting in a 'hit' when the dependency graph is inferred.

Experiments with the PhoneLabs data set demonstrate a very small increase in recall when the model incorporates the dependency graph. Similar effects can be seen for NDCG and HR. However, for $k = 5$, a significant decrease in precision can be observed, while values higher than $k = 10$ show a much smaller decrease in precision. Using the dependency graph as a post-processing step has a similar effect, with a minimal increase in recall, NDCG and HR, and a comparable drop in precision. Interestingly, for $k = 5$ the drop in precision is smaller than when the dependency graph is incorporated in the model.

For the IoT data set, it can be seen that the use of the dependency graph has a significant effect on model performance. Large improvements can be seen in recall when using the dependency graph, at the cost of slightly lower precision. Using the dependency graph as a post-processing step tends to have a less negative impact on precision, NDCG and HR compared to incorporating it in the model directly. The PhoneLabs dataset is less affected by the use of the dependency graph, indicating that predictions tend to be rather 'complete', as not many (ir)relevant elements are added to the prediction and metrics remain similar. Thus, the effect of the dependency graph differs between data sets, but could potentially have a significant positive effect on model performance for some databases.

The difference between incorporating the dependency graph directly into the model and using it as a post-processing step is notable. It can be observed that incorporating the dependency graph directly into the model has a negative impact on precision and that using the graph only as a post-processing step is arguably more effective in terms of recall, NDCG, and HR than not using the dependency graph for all values of k. The reason for this is that when predicting the next sets recurrently, the dependency graph being incorporated in the model could introduce more errors into the input of the next time-step. As there are always errors in each prediction, the dependency graph compounds these errors which could affect the performance of the model.

6.3 Effect of the Repeated Element Component

Experiments were also performed to determine the effect of the repeated element component γ, as described in Section 4.1.3. We compare the performance of the model with this component, i.e.

$$o(\hat{\mathbf{v}}_t) = \mathrm{softmax}(W_o s_t \circ (\mathbf{1} - \beta \circ \mathbf{w}) + \gamma \circ \mathbf{w}),$$

to that without it, i.e.

$$o(\hat{\mathbf{v}}_t) = \mathrm{softmax}(W_o s_t).$$

Both implementations of the model make use of the dependency graph as a post-processing step and use input sequences of length $l = 20$, and output sequence lengths of $m = 2$. Figures 6.9-6.16 show how this component affects the recall, precision, NDCG, and HR scores. For more detailed results, the reader is referred to Appendix B.4. As can be seen in Figures 6.9-6.16, the repeated element component γ improves recall, precision, and HR scores for all values of k in both the IoT and PhoneLabs data sets. In the PhoneLabs data set, NDCG is slightly better for all values of k, but for the IoT data sets $k = 15$ and $k = 20$ slightly worsen. Overall, it can be said that the repeated element component improves the model performance, and confirms our assumption that this component can aid in modeling the relation between partial QEPs occurring in the past and those occurring in the future.

6.4 Effect of the Sequence Lengths

The effect of longer input and output sequence lengths was examined by comparing model performance with $m = 2$ and values $l \in \{20, 30, 40\}$, and that with $l = 20$ and values $m \in \{2, 3, 5\}$. Figures 6.17-6.24 summarize the findings.

As can be seen in Figures 6.18, 6.20, 6.22 and 6.24, input sequence lengths of $l = 30$ and $l = 40$ perform significantly worse than $l = 20$ for all metrics and the IoT data set. These results lead us to believe that the model has difficulties modeling the temporal relation of longer input sequences. However, the difference between $l = 20$ and $l = 30$ is smaller than the difference between $l = 30$ and $l = 40$. In addition, NDCG is slightly better for $l = 40$ than $l = 30$. Hence, it is difficult to say what the actual effect of input sequence length is, and more experiments should be performed with a wider range of sequence lengths. For the PhoneLabs data set, no notable difference was observed. Thus, the effect of input sequence lengths differs between data sets. This indicates that potentially larger input sequences could be exploited to model longer temporal relations in some data sets.

Experiments were also performed when predicting $m = 3$ and $m = 5$ subsequent sets into the future. Predicting the future sets recurrently can always introduce errors at each time step. When that happens, the temporal correlation between sets is also affected. Therefore, it is expected that the performance of the model degrades the more sets are predicted into the future. Nevertheless, the usefulness of the model could be improved the more sets into the future can reliably be predicted. Figures 6.17, 6.19, 6.21 and 6.23 show the different metrics for both data sets for different values of m. It can be observed that all metrics decrease for both the IoT and PhoneLabs data sets when the number of subsequent sets to predict is increased. However, for the PhoneLabs data set, the decrease is less prominent. Hence, it depends on the data set whether predicting more queries into the future is worth the lesser performance, and the parameter m can best be determined by the user of the model.

6.5 Effect of the Partial QEP Size

Until know, we have only studied the model's performance in terms of an average metric across the m predicted sets. That is, the predicted sets were considered in its entirety without making a distinction based on the type of elements in those sets. However, a clear categorization of partial QEPs exists based on their size, which corresponds to the number of operators/nodes in its tree. Only being able to predict partial QEPs with a single operator is analogous to predicting which tables and indexes are queried. While this could be considered a feat in itself, as accurately predicting which table(s) will be queried next is not trivial, the opportunities for optimization are limited. Since it fails to predict computationally expensive operators, such as joins, its main use would be in reducing disk I/Os. Therefore, we also investigate the model's performance based on the size of the predicted partial QEPs.

The previous experiments used various metrics to measure performance. However, precision, HR, and NDCG are arguably less meaningful when we focus on only a subset of the partial QEPs in the ground-truth and predicted sets. Thereby, we will focus on recall for this experiment, and investigate the fraction of partial QEPs of each size present in the ground-truth set that the model is able to correctly predict. Note that we only calculate recall when a particular QEP size is actually present in the ground-truth set, to avoid skewing the results based on frequency.

Figures 6.25 through 6.32 visualize the recall scores by data set, partial QEP size, evaluation method, and the number of predicted elements per set k. For more detailed results, the reader is again referred to Appendix B.5. Looking at Figures 6.25-6.28, the performance of all methods decreases for larger partial QEP sizes in the IoT data set. This implies that as the complexity of a partial QEP increases, it becomes more difficult to predict. Furthermore, limitations of the other methods become apparent when compared to the model. Where the model is able to achieve recall values around 25% for partial QEP with up to four operators, the other methods degrade more quickly and fail to make accurate predictions for partial QEPs with more than two operators. In particular, the model's recall is more than double that of other methods for all partial QEP sizes larger than two.

Predicting more elements for each subsequent set has little effect on the recall relative to the partial QEP size except for one particular case. When looking at the TopKFrequent method, one can observe double the recall score for partial QEPs with a single operator when $k = 10$ compared to $k = 5$. Additionally, it outperforms the model for partial QEPs with a single operator for all $k \in \{10, 15, 20\}$, indicating that it is actually better than the model for predicting the tables or indexes involved in a query. One explanation for this is that the partial QEPs with a single operator are also among the most frequent, and at least one of which is present in every query. As such, the TopKFrequent method, which predicts the most frequent partial QEPs, is able to obtain a high recall score.

For the PhoneLabs data set, Figures 6.29-6.32 visualize the results. As can be seen, the graphs are similar across methods and values of k. This indicates that all predictive methods have difficulty predicting partial QEPs with more than two operators. A logical explanation for this would be the lack of larger size QEPs in the data set.

6.6 The Model in a Practical Setting

Each method described in Section 5.5.2 was run in a simulation with different cache capacities and compared to the 'none' method where no caching mechanism is used. This none method serves as a baseline for our evaluation and simply computes the result of every query template in the testing set as they arrive, so cache capacity has no effect on its performance. The test set of the IoT data set consists of exactly 21898 queries, and computing all the queries took a total of 164774 ms, approximately 36 minutes. As such, the average time to evaluate a query was found to be 98.86 ms. The results of the model's experimental performance are summarized in Table 6.5 and Figure 6.33.

As can be seen in Table 6.5 and Figure 6.33, the use of an LRU cache with 25 MB of capacity reduces the absolute time to process all queries to 1807033 ms, roughly 30 minutes. Consequently, the average time per query is reduced to 82.43 ms, resulting in a total speedup of 1.20x. When the cache capacity is increased, the results of more partial QEPs can remain in cache and we expect the number of cache hits to increase. The results demonstrate this, as doubling the cache capacity to

IoT	$n = 21898, N = 5801, l = 20, k = 5, m = 2$, with graph					
	25 MB			50 MB		
Method	**Time (ms)**	**Avg. Time (ms)**	**Speedup**	**Time (ms)**	**Avg. Time (ms)**	**Speedup**
None	2164774	98.86	1.00	2164774	98.86	1.00
LRU	1807033	82.43	1.20	1441648	65.76	1.50
FIFO	1807013	82.43	1.20	1312225	59.86	1.65
LRU + m	1790051	81.74	1.21	1275738	58.26	1.70
FIFO + m	1791935	81.83	1.21	1271156	58.05	1.70
Model	**189606**	**8.66**	**11.42**	**175791**	**8.03**	**12.31**
	75 MB			100 MB		
	Time (ms)	**Avg. Time (ms)**	**Speedup**	**Time (ms)**	**Avg. Time (ms)**	**Speedup**
None	2164774	98.86	1.00	2164774	98.86	1.00
LRU	1123194	51.23	1.93	1058300	48.27	2.05
FIFO	1110006	50.63	1.95	1036973	47.30	2.09
LRU + m	931942	42.56	2.32	892023	40.74	2.43
FIFO + m	910689	41.59	2.38	810671	37.02	2.67
Model	**157596**	**7.20**	**13.74**	**145250**	**6.63**	**14.90**

Table 6.5: Query answering time with different methods for the **IoT** data set

50 MB results a 1.5x speedup of query answering time. Analogously, increasing the cache capacity to 75 and 100 MB yields speedups of 1.93x and 2.05x respectively. Similar results can be seen for the FIFO caching policy, except that it slightly outperforms the LRU cache when the capacity is 50 MB or higher. The reason for the FIFO policy outperforming LRU is suspected to be due to frequent partial QEP corresponding to the base table operators, such as table scans. The results of these partial QEP are among the biggest in byte size, and thus occupy large amounts of memory with relatively little computation time saved when used from cache. Since the FIFO policy makes no distinction between elements based on cache hits, a slight performance increase can be seen. Overall, the results demonstrate that query speed can be improved by 20% up to 100% using relatively simple caching policies, a reasonable cache size, and the encoding scheme presented in Chapter 3.

Unfortunately, a selective caching strategy based on model predictions suffers from the fact that partial QEPs need to remain in memory before deciding what can be evicted. Therefore, the potential gain in reduced memory size by only caching what is predicted is limited. Nevertheless, the eviction of partial QEP results before they are needed can be avoided in some cases by forcing the results of predicted partial QEPs to remain cached until the arrival of the corresponding query. Hence, we also considered an alteration to the LRU and FIFO caching policies, making them 'smarter' by use of the model. For this we assume that there is enough time in between query arrivals to infer the model for the next m queries. In particular, the model was used to selectively avoid the eviction of results of predicted partial QEPs. The LRU/FIFO with model (+ m) strategies test the extent to which this can be used to improve query performance. The results of the LRU/FIFO with model strategies can also be seen in Table 6.5 and Figure 6.33. As can be seen, the performance is almost the same for a cache size of 25 MB. But, when the cache capacity is increased, the difference becomes apparent. For cache sizes of 50 MB or higher, selectively keeping the results of partial QEPs cached based on the model's predictions yields a significant improvement in query answering time. Specifically, combining the model with LRU and FIFO caching policies results in speedups ranging from 1.20 - 2.67x depending on the cache capacity. In general, the results show that the model can be used to improve upon LRU and FIFO caching policies to improve query answering time.

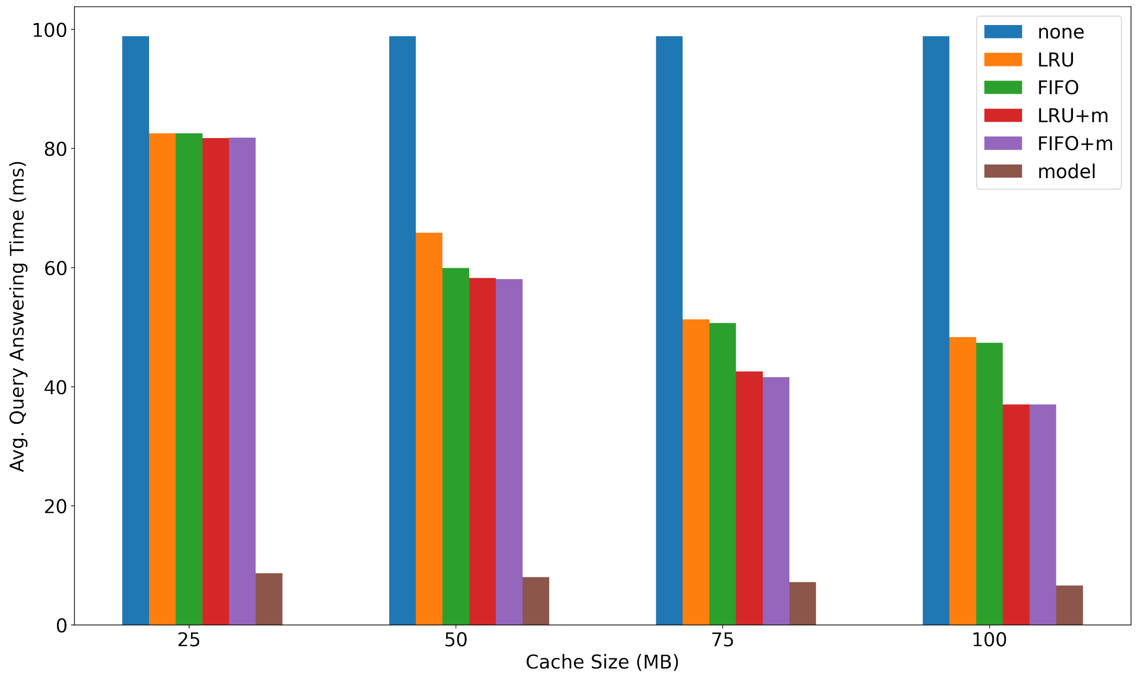


Figure 6.33: Average query answering time with different methods by cache size for the **IoT** data set

Lastly, the model was tested autonomously. For this we also assume that there is enough time in between query arrivals to infer the model for the next *m* queries, and subsequently compute the result preemptively and cache it in memory. In other words, this strategy aims to measure to what extent a DBMS could be optimized if the model was used to its fullest potential. The results in Table 6.5 and Figure 6.33 demonstrate that the average time to answer a query can be reduced from 98.86 ms down to 8.66 ms. As such, the total answering time of evaluating the 21898 queries is reduced from approximately 36 minutes to a mere 3 minutes. This corresponds to a vast speedup of 11.30x. In addition, this indicates that a smaller cache is needed to achieve high performance, unlike LRU and FIFO methods (both with or without model usage), which are significantly more dependent on the cache capacity. Even better performance can be observed for larger cache sizes, with the average query answering time reaching as low as 6.63 ms, a speedup of 14.90x. However, it must be noted that these results hold only under the assumption that predictions and pre-computation can be performed before the query arrivals. In database systems, queries can also arrive in rapid succession, such that they can't benefit from the model's predictions. Therefore, these results are only representative of queries that arrive with a delay big enough for model inference and pre-computation.

6.7 Computation Time

Query Encoding

A distinction can be made between creating a usable data set, and only encoding a QEP with an established encoding scheme. The prior includes the parsing, pre-processing, optimization and encoding of the queries. On the other hand, the latter corresponds to only the additional overhead that is caused by using the encoding scheme, given that the other steps need to be performed regardless of using the encoding scheme. The query encoding was performed on the same machine as the cost calculation, a desktop PC with a 3.5 GHz Quad-Core 64-bit CPU architecture, 16 GB of RAM and with the data stored on an SSD that supports 94.000 I/O operations per second.

For the IoT data set, encoding the 111087 queries into a data set, as per the encoding scheme outlined in Chapter 3, took approximately 31 minutes. The PhoneLabs data set took significantly longer because of the larger number of queries. In total, encoding the 553751 queries took approximately 271 minutes, or 4.5 hours.

However, with an established mapping between partial QEPs and indices, encoding a query takes approximately 1 ms on average for both data sets. Hence, if the model was to be used in practice, queries would have to be encoded as they arrive. This results in an additional overhead of 1 ms in processing time per query.

Model Training

Training of the model was performed on a 'Standard_NC6_Promo' virtual machine on the Microsoft Azure platform because of its GPU support. This virtual machine utilizes a 6-core CPU, 56 GB of RAM, and an Nvidia Tesla K80 GPU with 24GB of memory. The model was implemented to use Nvidia CUDA for parallel computing. Training the model from a fresh start for 10 epochs took approximately 6 hours, so about 35 minutes per epoch. However, since the model would converge fast — the best performance was generally observed within the first 5 epochs — training the model for best performance can be achieved in about 3 hours. Since the training and test sets were the same size for both the IoT and the PhoneLabs data set, no notable difference in training or test speed was observed.

Model Inference

Given that the main motivation for this work is to provide a means of improving query performance, we require inference to be fast. Depending on the database, queries can rapidly arrive in succession during bursty workloads. Especially queries that are automatically generated by some process may arrive within less than a thousandth of a millisecond from one another.

For inference, we used the same virtual machine as used during training. With the model loaded in memory and utilizing the dependency graph, we observed that inference takes approximately 32 ms on average. The minimum and maximum inference time we observed were 24 and 157 ms respectively. Without the dependency graph, inference takes approximately 29 milliseconds on average, with extremes being 22 and 153 ms. Thus, the effect of the dependency graph on the inference time is small and the input sequence has the biggest impact on inference time.

Note that inference time can be reduced significantly with better hardware. Upgrading the GPU in the virtual machine to an Nvidia Tesla T4 reduces inference time to approximately 21 ms.

Chapter 7

Conclusion

7.1 Overview

First of all, this work addressed the feature engineering of SQL queries. In particular, we investigated how the dimensionality of SQL queries could be reduced such that query patterns could be exploited and meaningful predictions could be made. To this end, the concept of query execution plan (QEP) fingerprinting was introduced. This encoding scheme is based on the QEP of a query in order to capture the reusable elements, i.e. the intermediate results, of a query in a low-dimensional representation. In addition, transforming each query into a set of elements accommodates partially correct predictions, which could increase a predictive model's potential. Consequently, predictions of these intermediate query results can be used to optimize a database management system (DBMS) through caching, pre-emptive computation and more. Thus, a framework implementing this encoding scheme was developed. This framework was used to transform two real-world query logs into data sets that are suitable for machine learning techniques and other predictive methods.

Secondly, a deep neural architecture was presented to model the joint distribution between encoded *sequences* of queries. This model architecture enables the prediction of partial QEPs included in future queries based on the most recently seen sequence of queries. Furthermore, this model was fitted with several components to improve its performance, such as a dependency graph of the partial QEPs, and a repeated element component to explicitly model the past element to future element temporal relation. In addition, we demonstrate how a greedy inference algorithm could be used to infer the model in approximately 30ms or less.

Subsequently, experiments studied the performance of the model and the effect of the individual components on two data sets. First, the model was evaluated with a set of commonly used metrics and compared to several other prediction methods. The results demonstrate that good performance and generalization can be achieved for both databases. For the Internet of Things data set, the model outperforms the best performance of other methods with respect to recall by 46.8%-51.9%. However, in the PhoneLabs data set, the model only outperforms other methods by 2.2-3.7% with respect to recall. Additionally, results show that increasing the number of elements that are predicted at each subsequent set negatively affects the precision of the model, demonstrating a limitation of the model compared to other methods.

Further analysis examined the effect of the dependency graph. Results show that incorporating the dependency graph in the model directly improves recall scores at the cost of lower precision, normalized discounted cumulative gain (NDCG) and hit rate (HR). On the other hand, using the dependency graph as a post-processing step has a net-positive effect on the predictive performance. In addition to the higher recall, higher NDCG and HR scores could be observed with only slightly lower precision scores. Furthermore, the effect of the repeated element component γ was examined, by comparing two distinct output functions of the Gated Recurrent Unit (GRU) cell. In general, the effect is positive, with slightly higher scores across all metrics and the number of predicted elements for each subsequent set k, with the exception of NDCG scores for $k = 15$ and $k = 20$. Moreover, increasing the number of subsequent queries m for which to predict the set of partial QEPs has a negative effect on all metrics for both data sets. Similar performance degradation can be seen when the input sequence length l is increased.

The usefulness of the model was further examined by placing it in a practical setting. In particular, a simulation was created where the encoded queries would arrive sequentially and a cache of fixed memory size was available. The results show that simple caching strategies such as LRU and FIFO, and relatively small cache sizes can be used to improve the answering time of queries encoded with QEP fingerprinting by up to 2.09x. When using the model's predictions to improve upon these caching policies enables speedups of up to 2.67x were observed. Additionally, in the scenario that queries can be pre-computed based on the model's predictions, speedups of up to 14.90x could be achieved.

In summary, this work proposes a novel method to encode queries into vectors of usable dimensions that can be used to model the temporal correlations between queries in a DBMS. The proposed encoding scheme makes it suitable for use in different DBMSs with different schemas and query workloads, and enables optimization of query processing through caching or prediction. In addition, experiments demonstrate how a deep learning framework can be used with our encoding scheme to build a predictive model for SQL query data and exploit query patterns and consequently optimize query processing in a DBMS.

7.2 Limitations

The first limitation to be addressed is that the encoding scheme and model are not schema-agnostic. Especially in less mature databases, schema changes can happen frequently. As a consequence, queries result in different plans, so also in different query encodings. Not only could this invalidate query encodings generated prior to a schema change, but encoding future queries could result in partial QEPs that were never seen before by the model. Even though the model could still be used in this case, the modeled temporal correlation between past and future queries becomes less effective. Furthermore, predictions of partial QEPs could now be invalid, i.e. correspond to plans that can't be executed on the database. Thus, it would be necessary to re-encode the training set and re-train the model for every schema change, which is computationally intensive and unpractical for databases with frequent schematic updates. Thus, the practicality of a deep learning framework for query optimization is limited to databases with infrequent schema changes.

Furthermore, while the use of query templates reduces the dimensionality of the encoded vectors, predictions are less useful as filter parameters are not accounted for. Therefore, the results of partial QEPs that were correctly predicted don't correspond to the results of the query exactly, and more post-processing is needed. Even though the results demonstrate that a high amount of optimization could be achieved, it is unknown to what extent this optimization will be affected by the additional post-processing. More experimentation is needed to determine the difference between evaluating a QEP generated from a *query*, and one generated from a *query template*, where post-processing is performed to obtain the query result in the latter. However, as was argued prior in Section 3.1.2, we expect this effect to be small. Given that the inclusion of additional table attributes in projections and applying filters retroactively would likely not have a high overhead compared to the overall processing time of the QEP.

Moreover, using fixed input and output sequence lengths as training samples makes it difficult to model long-term patterns. For instance, a database may experience a similar query load daily at a specific time, or at the end of the month. An example would be scheduled cron[1] jobs that run automated tasks such as sanity tests and batch operations periodically at a specified time. By fixing the sequence length to l, we can only model short-term patterns that don't span more than l queries, and these types of scheduled query loads cannot be modeled reliably. Even though parameters l and

[1]cron is a time-based task scheduler commonly used in Unix-based systems

m can be chosen by the user such that the desired patterns can be modeled, very large values of l are not practical. The reason for this is that the loss needs to be backpropagated through every recurrent unit in the sequence, so training times become restrictive. As a result, encoder-decoder RNN models are hard to scale to very long sequences [33], [45].

Finally, inference time could be considered problematic. With an experimental inference time of 21 ms, queries arriving with less than that in between are unable to benefit from the model, even without accounting for the cost of pre-computing the predictions. Yet, many DBMSs experience heavy loads with queries arriving in rapid succession, with often only a fraction of a millisecond in between arrivals. In this case, the usefulness of the model is limited, as predictions can't be made fast enough to enable any kind of optimization. Therefore, the model is best suited for applications where queries arrive somewhere in the range of 50ms or more between each other, and enough time is available to infer the model and pre-compute a set of partial QEPs. However, the 21 ms corresponds to inference with an Nvidia Tesla T4 GPU. Potentially much smaller inference times can be achieved with the commonly used state-of-the-art Nvidia Tesla V100 GPU, but this was not available to us for testing. Alternatively, systems with heavy loads could implement a delayed start feature, where the model would only be inferred when a certain amount of time has passed since the last query arrival.

7.3 Future Work

First of all, more experimentation could be performed using the presented encoding scheme and deep learning-based model. For instance, different data sets, different input/output sequence lengths, and altered model architectures could reveal interesting results and potentially improve predictive performance. Since we only performed experiments with the top-1 QEP of a query, as was generated by the query optimizer, experimentation with multiple QEPs per query could also be conducted. For example, experiments could investigate the effect of considering multiple QEPs as input, and compare the different methods of handling these, such as considering the top-k frequent partial QEPs or taking the union over each set of partial QEPs. In turn, this could enable the optimization of future queries to a greater extent.

Results indicate that the model could be used to optimize LRU/FIFO caching policies and vastly reduce query answering times by pre-computing predicted partial QEPs. However, the partial QEPs correspond to the QEPs of query templates. Thereby, filter parameters were replaced with placeholders and predictions don't always correspond to the results of the query exactly. Therefore, future work could perform experiments without transforming queries into query templates. Alternatively, QEPs could be altered in the encoding such that filter attributes are included in projections and that filtering can still be applied retroactively to obtain the query result. Unfortunately, query parameters tend to contain sensitive information so obtaining a data set that includes filter parameters is difficult. As such, the cost of altering QEPs for this purpose and retroactively filtering the predictions was not included in the cost model that was presented in Chapter 5, so future experiments could investigate the extent to which this overhead affects the optimization potential. Perhaps more interesting is fully integrating the model with a query optimizer. For instance, a query optimizer could be extended such that the use of predicted partial QEPs is maximized. It could subtract the cost of cached partial QEPs from the QEP candidates of a query and select the best QEP based on the new cost estimates.

Future work could further examine the exploitation of query patterns in database systems based on QEP fingerprinting. Previous research demonstrates how deep reinforcement learning can be applied to query optimization [1]–[6] and query performance estimation [7], but no research addresses the exploitation of query patterns with deep reinforcement learning. Even though we have shown how deep learning can be used to induce a predictive model for SQL query data, approaches purely based on deep learning models have certain limitations. For instance, deep reinforcement learning could address the problem of schema dependence and alleviate the need of training a model before deployment. With its schema-agnostic and 'on-line' learning properties, it could potentially provide a solution that requires less setup time and is more adaptive to changes than deep learning methods.

Furthermore, the research presented in this work focused on database query patterns. Future work could investigate user-specific or location-specific query patterns by associating queries to a source. For example, this work was based on the idea that a database experiences query patterns associated with an application, system or user and that the query log of the said database could exhibit these patterns. However, a database could also be subject to queries from multiple applications, systems, or users that each exhibit their own respective patterns. Even though it may be more difficult to model, and multiple instances of a model might be needed, temporal correlations between queries could be more significant. It might also be interesting to investigate databases with queries manually written by end-users, such as SDSS's SkyServer [41]. Moreover, models could be deployed on edge nodes in distributed database systems, where only the patterns of queries going through a particular edge node are exploited. Experiments in distributed database systems could also study the potential gain when query results are prepared for on an edge node and don't have to be retrieved from a data center.

Finally, future work could investigate the modeling of longer-term relations. Unfortunately, scaling sequence to sequence model to very long sequences is difficult using backpropagation through time (BPTT). In addition, gradient descent becomes increasingly inefficient when the temporal span of the dependencies increases [53]. However, various approaches exist for modeling longer sequences. First, Trinh et al. [45] propose the use of an unsupervised auxiliary loss to enable learning with only a few BPTT steps from the supervised loss. In addition, they argue that it is applicable to online learning systems or those that process very long sequences. It would be interesting to investigate whether such an auxiliary loss could be incorporated in our model and potentially enable the modeling of longer sequences. Alternatively, different architectures could be considered that better scale to longer sequences. In particular Convolutional Neural Networks (CNN) have been proposed as an alternative to encoder-decoder models for sequence to sequence prediction [54], [55]. They achieve impressive results with a conceptually simpler model that has fewer parameters to train and does not use BPTT, potentially allowing it to scale better to longer sequences.

References

[1] S. Krishnan, Z. Yang, K. Goldberg, J. M. Hellerstein, and I. Stoica, “Learning to optimize join queries with deep reinforcement learning,” *CoRR*, vol. abs/1808.03196, 2018. arXiv: 1808.03196. [Online]. Available: http://arxiv.org/abs/1808.03196.

[2] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil, “LEO - db2’s learning optimizer,” in *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, Eds., Morgan Kaufmann, 2001, pp. 19–28. [Online]. Available: http://www.vldb.org/conf/2001/P019.pdf.

[3] J. Heitz and K. Stockinger, “Join query optimization with deep reinforcement learning algorithms,” *CoRR*, vol. abs/1911.11689, 2019. arXiv: 1911.11689. [Online]. Available: http://arxiv.org/abs/1911.11689.

[4] R. C. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul, “Neo: A learned query optimizer,” *Proc. VLDB Endow.*, vol. 12, no. 11, pp. 1705–1718, 2019. DOI: 10.14778/3342263.3342644. [Online]. Available: http://www.vldb.org/pvldb/vol12/p1705-marcus.pdf.

[5] B. Glasbergen, M. Abebe, K. Daudjee, S. Foggo, and A. Pacaci, “Apollo: Learning query correlations for predictive caching in geo-distributed systems,” in *Proceedings of the 21st International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*, M. H. Böhlen, R. Pichler, N. May, E. Rahm, S.-H. Wu, and K. Hose, Eds., OpenProceedings.org, 2018, pp. 253–264. DOI: 10.5441/002/edbt.2018.23. [Online]. Available: https://doi.org/10.5441/002/edbt.2018.23.

[6] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska, “Bao: Learning to steer query optimizers,” *CoRR*, vol. abs/2004.03814, 2020. arXiv: 2004.03814. [Online]. Available: https://arxiv.org/abs/2004.03814.

[7] R. C. Marcus and O. Papaemmanouil, “Plan-structured deep neural network models for query performance prediction,” *Proc. VLDB Endow.*, vol. 12, no. 11, pp. 1733–1746, 2019. DOI: 10.14778/3342263.3342646. [Online]. Available: http://www.vldb.org/pvldb/vol12/p1733-marcus.pdf.

[8] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi, “Learning state representations for query optimization with deep reinforcement learning,” in *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning, DEEM@SIGMOD 2018, Houston, TX, USA, June 15, 2018*, S. Schelter, S. Seufert, and A. Kumar, Eds., ACM, 2018, 4:1–4:4. DOI: 10.1145/3209889.3209890. [Online]. Available: https://doi.org/10.1145/3209889.3209890.

[9] J. R. David Reinsel John Gantz, “Data age 2025, the digitization of the world from edge to core,” International Data Corporation, Tech. Rep., 2018.

[10] D. Bell, *Distributed database systems*. Addison-Wesley Longman Publishing Co., Inc., 1992.

[11] I. T. Bowman and K. Salem, "Semantic prefetching of correlated query sequences," in *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, R. Chirkova, A. Dogac, M. T. Özsu, and T. K. Sellis, Eds., IEEE Computer Society, 2007, pp. 1284–1288. DOI: 10.1109/ICDE.2007.368994. [Online]. Available: https://doi.org/10.1109/ICDE.2007.368994.

[12] M. Casciaro, *Node.js design patterns : get the best out of Node.js by mastering a series of patterns and techniques to create modular, scalable, and efficient applications*. Birmingham: Packt Publishing, 2016, ISBN: 9781785885587.

[13] Google. (2017). "Google's edge network," [Online]. Available: https://peering.google.com/infrastructure.

[14] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds., pp. 3104–3112, 2014. [Online]. Available: https://proceedings.neurips.cc/paper/2014/hash/a14ac55a4f27472c5d894ec1c3c743d2-Abstract.html.

[15] W. Chan, N. Jaitly, Q. V. Le, and O. Vinyals, "Listen, attend and spell: A neural network for large vocabulary conversational speech recognition," pp. 4960–4964, 2016. DOI: 10.1109/ICASSP.2016.7472621. [Online]. Available: https://doi.org/10.1109/ICASSP.2016.7472621.

[16] D. Bahdanau, J. Chorowski, D. Serdyuk, P. Brakel, and Y. Bengio, "End-to-end attention-based large vocabulary speech recognition," in *2016 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2016, Shanghai, China, March 20-25, 2016*, IEEE, 2016, pp. 4945–4949. DOI: 10.1109/ICASSP.2016.7472618. [Online]. Available: https://doi.org/10.1109/ICASSP.2016.7472618.

[17] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1409.0473.

[18] *Engines ranking*, Jul. 2021. [Online]. Available: https://db-engines.com/en/ranking.

[19] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, 2016. DOI: 10.1145/2934664. [Online]. Available: http://doi.acm.org/10.1145/2934664.

[20] T. A. S. Foundation, *Apache spark sql*. [Online]. Available: https://spark.apache.org/sql.

[21] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, no. 6, pp. 377–387, 1970. DOI: 10.1145/362384.362685. [Online]. Available: http://doi.acm.org/10.1145/362384.362685.

[22] B. Schwartz, *High performance MySQL*. Sebastopol, Calif: O'Reilly Media, 2008, ISBN: 9780596101718.

[23] C. Li, K. C.-C. Chang, I. F. Ilyas, and S. Song, "Ranksql: Query algebra and optimization for relational top-k queries," F. Özcan, Ed., pp. 131–142, 2005. DOI: 10.1145/1066157.1066173. [Online]. Available: https://doi.org/10.1145/1066157.1066173.

[24] R. Marcus and O. Papaemmanouil, "Deep reinforcement learning for join order enumeration," in *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2018, Houston, TX, USA, June 10, 2018*, R. Bordawekar and O. Shmueli, Eds., ACM, 2018, 3:1–3:4. DOI: 10.1145/3211954.3211957. [Online]. Available: https://doi.org/10.1145/3211954.3211957.

[25] D. Duvenaud, D. Maclaurin, J. Aguilera-Iparraguirre, R. Gómez-Bombarelli, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams, "Convolutional networks on graphs for learning molecular fingerprints," in *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds., 2015, pp. 2224–2232. [Online]. Available: `https://proceedings.neurips.cc/paper/2015/hash/f9be311e65d81a9ad8150a60844bb94c-Abstract.html`.

[26] C. Bishop, *Pattern recognition and machine learning*. New York: Springer, 2006, ISBN: 978-0-387-31073-2.

[27] F. Chollet, *Building powerful image classification models using very little data*, Jun. 2016. [Online]. Available: `https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html`.

[28] R. M. Castro and R. D. Nowak, *Statistical Learning Theory, A Gentle Primer*. 2019, Early draft.

[29] V. N. Vapnik, *The Nature of Statistical Learning Theory*. Springer New York, 2000. DOI: `10.1007/978-1-4757-3264-1`. [Online]. Available: `https://doi.org/10.1007/978-1-4757-3264-1`.

[30] A. Amini, A. Soleimany, S. Karaman, and D. Rus, "Spatial uncertainty sampling for end-to-end control," *CoRR*, vol. abs/1805.04829, 2018. arXiv: `1805.04829`. [Online]. Available: `http://arxiv.org/abs/1805.04829`.

[31] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," in *Neurocomputing: Foundations of Research*. Cambridge, MA, USA: MIT Press, 1988, pp. 696–699, ISBN: 0262010976.

[32] M. I. Jordan, "Chapter 25 - serial order: A parallel distributed processing approach," in *Neural-Network Models of Cognition*, ser. Advances in Psychology, J. W. Donahoe and V. Packard Dorsel, Eds., vol. 121, North-Holland, 1997, pp. 471–495. DOI: `https://doi.org/10.1016/S0166-4115(97)80111-2`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0166411597801112`.

[33] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, pp. 1735–80, Dec. 1997. DOI: `10.1162/neco.1997.9.8.1735`.

[34] K. Cho, B. van Merrienboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, A. Moschitti, B. Pang, and W. Daelemans, Eds., ACL, 2014, pp. 1724–1734. DOI: `10.3115/v1/d14-1179`. [Online]. Available: `https://doi.org/10.3115/v1/d14-1179`.

[35] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds., pp. 3104–3112, 2014. [Online]. Available: `https://proceedings.neurips.cc/paper/2014/hash/a14ac55a4f27472c5d894ec1c3c743d2-Abstract.html`.

[36] K. Cho, B. van Merrienboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, A. Moschitti, B. Pang, and W. Daelemans, Eds., ACL, 2014, pp. 1724–1734. DOI: `10.3115/v1/d14-1179`. [Online]. Available: `https://doi.org/10.3115/v1/d14-1179`.

[37] O. Vinyals, S. Bengio, and M. Kudlur, "Order matters: Sequence to sequence for sets," in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2016. [Online]. Available: `http://arxiv.org/abs/1511.06391`.

[38] H. Hu and X. He, "Sets2sets: Learning from sequential sets with neural networks," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*, A. Teredesai, V. Kumar, Y. Li, R. Rosales, E. Terzi, and G. Karypis, Eds., ACM, 2019, pp. 1491–1499. DOI: 10.1145/3292500.3330979. [Online]. Available: https://doi.org/10.1145/3292500.3330979.

[39] A. Vogelsgesang, M. Haubenschild, J. Finis, A. Kemper, V. Leis, T. Mühlbauer, T. Neumann, and M. Then, "Get real: How benchmarks fail to represent the real world," in *Proceedings of the 7th International Workshop on Testing Database Systems, DBTest@SIGMOD 2018, Houston, TX, USA, June 15, 2018*, A. Böhm and T. Rabl, Eds., ACM, 2018, 1:1–1:6. DOI: 10.1145/3209950.3209952. [Online]. Available: https://doi.org/10.1145/3209950.3209952.

[40] P. A. Boncz, T. Neumann, and O. Erling, "TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark," in *Performance Characterization and Benchmarking - 5th TPC Technology Conference, TPCTC 2013, Trento, Italy, August 26, 2013, Revised Selected Papers*, R. Nambiar and M. Poess, Eds., ser. Lecture Notes in Computer Science, vol. 8391, Springer, 2013, pp. 61–76. DOI: 10.1007/978-3-319-04936-6_5. [Online]. Available: https://doi.org/10.1007/978-3-319-04936-6%5C_5.

[41] *What is the sloan digital sky survey?* [Online]. Available: http://skyserver.sdss.org/dr16/en/sdss/sdsshome.aspx.

[42] O. Kennedy, J. A. Ajay, G. Challen, and L. Ziarek, "Pocket data: The need for TPC-MOBILE," in *Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things - 7th TPC Technology Conference, TPCTC 2015, Kohala Coast, HI, USA, August 31 - September 4, 2015. Revised Selected Papers*, R. Nambiar and M. Poess, Eds., ser. Lecture Notes in Computer Science, vol. 9508, Springer, 2015, pp. 8–25. DOI: 10.1007/978-3-319-31409-9_2. [Online]. Available: https://doi.org/10.1007/978-3-319-31409-9%5C_2.

[43] J. Laskowski. (2020). "The internals of spark sql (apache spark 2.4.5)," [Online]. Available: https://jaceklaskowski.gitbooks.io/mastering-spark-sql/content/.

[44] N. V. Chawla, N. Japkowicz, and A. Kotcz, "Editorial: Special issue on learning from imbalanced data sets," *SIGKDD Explor.*, vol. 6, no. 1, pp. 1–6, 2004. DOI: 10.1145/1007730.1007733. [Online]. Available: https://doi.org/10.1145/1007730.1007733.

[45] T. H. Trinh, A. M. Dai, T. Luong, and Q. V. Le, "Learning longer-term dependencies in rnns with auxiliary losses," Proceedings of Machine Learning Research, vol. 80, J. G. Dy and A. Krause, Eds., pp. 4972–4981, 2018. [Online]. Available: http://proceedings.mlr.press/v80/trinh18a.html.

[46] M. Chancán and M. Milford, "Deepseqslam: A trainable CNN+RNN for joint global description and sequence-based place recognition," *CoRR*, vol. abs/2011.08518, 2020. arXiv: 2011.08518. [Online]. Available: https://arxiv.org/abs/2011.08518.

[47] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1412.6980.

[48] M. Grandini, E. Bagli, and G. Visani, "Metrics for multi-class classification: An overview," *CoRR*, vol. abs/2008.05756, 2020. arXiv: 2008.05756. [Online]. Available: https://arxiv.org/abs/2008.05756.

[49] M.-L. Zhang and Z.-H. Zhou, "A review on multi-label learning algorithms," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 8, pp. 1819–1837, 2014. DOI: 10.1109/TKDE.2013.39. [Online]. Available: https://doi.org/10.1109/TKDE.2013.39.

[50] Y. Wang, L. Wang, Y. Li, D. He, and T.-Y. Liu, "A theoretical analysis of NDCG type ranking measures," in *COLT 2013 - The 26th Annual Conference on Learning Theory, June 12-14, 2013, Princeton University, NJ, USA*, S. Shalev-Shwartz and I. Steinwart, Eds., ser. JMLR Workshop and Conference Proceedings, vol. 30, JMLR.org, 2013, pp. 25–54. [Online]. Available: http://proceedings.mlr.press/v30/Wang13.html.

[51] E. Christakopoulou and G. Karypis, "Local latent space models for top-n recommendation," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*, Y. Guo and F. Farooq, Eds., ACM, 2018, pp. 1235–1243. DOI: 10.1145/3219819.3220112. [Online]. Available: https://doi.org/10.1145/3219819.3220112.

[52] A. Dan and D. F. Towsley, "An approximate analysis of the LRU and FIFO buffer replacement schemes," in *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems, University of Colorado, Boulder, Colorado, USA, May 22-25, 1990*, G. J. Nutt, Ed., ACM, 1990, pp. 143–152. DOI: 10.1145/98457.98525. [Online]. Available: https://doi.org/10.1145/98457.98525.

[53] Y. Bengio, P. Y. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Trans. Neural Networks*, vol. 5, no. 2, pp. 157–166, 1994. DOI: 10.1109/72.279181. [Online]. Available: https://doi.org/10.1109/72.279181.

[54] M. Elbayad, L. Besacier, and J. Verbeek, "Pervasive attention: 2d convolutional neural networks for sequence-to-sequence prediction," in *Proceedings of the 22nd Conference on Computational Natural Language Learning, CoNLL 2018, Brussels, Belgium, October 31 - November 1, 2018*, A. Korhonen and I. Titov, Eds., Association for Computational Linguistics, 2018, pp. 97–107. DOI: 10.18653/v1/k18-1010. [Online]. Available: https://doi.org/10.18653/v1/k18-1010.

[55] S. Bai, J. Z. Kolter, and V. Koltun, "An empirical evaluation of generic convolutional and recurrent networks for sequence modeling," *CoRR*, vol. abs/1803.01271, 2018. arXiv: 1803.01271. [Online]. Available: http://arxiv.org/abs/1803.01271.

Appendix A

Source Code

All source code used is collected in one repository and can be found at: `https://github.com/792x/Deep-Learning-For-SQL-Operators`

A.1 Pre-Processing IoT data

The source code for preprocessing and anonimizing the IoT data can be found in the Datasets/IoT sub-folder of the repository: `https://github.com/792x/Deep-Learning-For-SQL-Operators/blob/master/Datasets/IoT`

A.2 Pre-Processing PhoneLabs data

The source code for preprocessing the PhoneLabs data can be found in the Datasets/PhoneLabs sub-folder of the repository: `https://github.com/792x/Deep-Learning-For-SQL-Operators/blob/master/Datasets/PhoneLabs`

A.3 Encoding Framework

The source code for the encoding framework written for Apache Spark with the Scala API can be found in Framework sub-folder of the repository:
`https://github.com/792x/Deep-Learning-For-SQL-Operators/blob/master/Framework`

A.4 Encoder-Decoder Model

The source code for the Jupyter notebook implementing the encoder-decoder model in PyTorch can be found in the Model sub-folder of the repository:
`https://github.com/792x/Deep-Learning-For-SQL-Operators/blob/master/Model`

A.5 Simulation

The source code for the Jupyter notebook implementing the simulation can be found in the Simulation sub-folder of the repository:
`https://github.com/792x/Deep-Learning-For-SQL-Operators/blob/master/Simulation`

Appendix B

Results

B.1 Model Evaluation of Different Methods for output sequence lengths $m = 3$ and $m = 5$

IoT	$N = 5801, l = 20, m = 3$, with graph							
	$k = 5$				$k = 10$			
Method	**Recall**	**Precision**	**NDCG**	**HR**	**Recall**	**Precision**	**NDCG**	**HR**
RandomSet	0.0007	0.0005	0.0011	0.0000	0.0018	0.0007	0.0026	0.0000
TopKFrequent	0.3061	0.1522	0.2607	0.1548	0.5156	0.1396	0.3470	0.7772
ExpectRepeat	0.3459	0.3483	0.3654	0.2838	0.3459	0.3483	0.3601	0.2838
Model	0.7487	0.3816	0.6141	0.5654	0.7688	0.2126	0.5764	0.6084
	$k = 15$				$k = 20$			
	Recall	**Precision**	**NDCG**	**HR**	**Recall**	**Precision**	**NDCG**	**HR**
RandomSet	0.0026	0.0006	0.0034	0.0000	0.0037	0.0007	0.0053	0.0000
TopKFrequent	0.5733	0.1117	0.3711	0.8856	0.6174	0.0992	0.3910	0.8856
ExpectRepeat	0.3459	0.3483	0.3594	0.2838	0.3459	0.3483	0.3594	0.2838
Model	0.7677	0.1443	0.5336	0.5496	0.8076	0.1176	0.5259	0.6264

Table B.1: Evaluation of different methods on the **IoT** data set with graph, $m = 3$

IoT	$N = 5801, l = 20, m = 5$, with graph							
	$k = 5$				$k = 10$			
Method	**Recall**	**Precision**	**NDCG**	**HR**	**Recall**	**Precision**	**NDCG**	**HR**
RandomSet	0.0008	0.0006	0.0013	0.0000	0.0016	0.0006	0.0022	0.0000
TopKFrequent	0.3086	0.1534	0.2625	0.0846	0.5155	0.1394	0.3476	0.6960
ExpectRepeat	0.3675	0.3683	0.38452	0.2406	0.3670	0.3683	0.3801	0.2406
Model	0.5765	0.2959	0.4540	0.2654	0.5689	0.1670	0.3799	0.1910
	$k = 15$				$k = 20$			
	Recall	**Precision**	**NDCG**	**HR**	**Recall**	**Precision**	**NDCG**	**HR**
RandomSet	0.0026	0.0007	0.0038	0.0000	0.0036	0.0007	0.005	0.0000
TopKFrequent	0.5739	0.1117	0.3719	0.8342	0.6165	0.09866	0.3912	0.8342
ExpectRepeat	0.3671	0.3683	0.3796	0.2406	0.3671	0.3683	0.3796	0.2406
Model	0.6316	0.1322	0.3955	0.1800	0.6852	0.1058	0.4013	0.2662

Table B.2: Evaluation of different methods on the **IoT** data set with graph, $m = 5$

PhoneLabs	$N = 4729, l = 20, m = 3$, with graph							
	$k = 5$				$k = 10$			
Method	**Recall**	**Precision**	**NDCG**	**HR**	**Recall**	**Precision**	**NDCG**	**HR**
RandomSet	0.0010	0.0006	0.0015	0.0000	0.0022	0.0006	0.0029	0.0000
TopKFrequent	0.8980	0.5388	0.8812	0.8936	0.9238	0.2771	0.8936	0.9084
ExpectRepeat	0.9274	0.9277	0.9282	0.8988	0.9274	0.9277	0.9281	0.8988
Model	0.9481	0.5550	0.9411	0.9244	0.9574	0.2832	0.9448	0.9362
	$k = 15$				$k = 20$			
	Recall	**Precision**	**NDCG**	**HR**	**Recall**	**Precision**	**NDCG**	**HR**
RandomSet	0.0028	0.0006	0.0038	0.0000	0.0038	0.0006	0.0053	0.0000
TopKFrequent	0.9429	0.1883	0.9005	0.9196	0.9485	0.1419	0.9023	0.9268
ExpectRepeat	0.9274	0.9277	0.9281	0.8988	0.9274	0.9277	0.9281	0.8988
Model	0.9632	0.1896	0.9451	0.9450	0.9663	0.1429	0.9469	0.9480

Table B.3: Evaluation of different methods on the **PhoneLabs** data set with graph, $m = 3$

PhoneLabs	$N = 4729, l = 20, m = 5$, with graph							
	$k = 5$				$k = 10$			
Method	**Recall**	**Precision**	**NDCG**	**HR**	**Recall**	**Precision**	**NDCG**	**HR**
RandomSet	0.0011	0.0007	0.0016	0.0000	0.0020	0.0006	0.0028	0.0000
TopKFrequent	0.8916	0.5350	0.8757	0.8660	0.9180	0.2754	0.8882	0.8802
ExpectRepeat	0.9139	0.9142	0.9145	0.8646	0.9139	0.9141	0.9145	0.8646
Model	0.9370	0.5431	0.9265	0.8892	0.9488	0.9060	0.9010	0.2643
	$k = 15$				$k = 20$			
	Recall	**Precision**	**NDCG**	**HR**	**Recall**	**Precision**	**NDCG**	**HR**
RandomSet	0.00328	0.0007	0.0046	0.0000	0.0044	0.0007	0.006	0.0000
TopKFrequent	0.9403	0.1877	0.8964	0.8938	0.9465	0.1415	0.8983	0.9028
ExpectRepeat	0.9139	0.9142	0.9145	0.8646	0.9138	0.9142	0.9145	0.8646
Model	0.9556	0.1810	0.9060	0.9138	0.9612	0.9199	0.9252	0.1416

Table B.4: Evaluation of different methods on the **PhoneLabs** data set with graph, $m = 5$

B.2 Model Evaluation of Different Methods for input sequence lengths $l = 30$ and $l = 40$

IoT	$N = 5801, l = 30, m = 2$, with graph							
	$k = 5$				$k = 10$			
Method	**Recall**	**Precision**	**NDCG**	**HR**	**Recall**	**Precision**	**NDCG**	**HR**
RandomSet	0.0007	0.0005	0.0011	0.0000	0.0021	0.0009	0.0031	0.0002
TopKFrequent	0.3123	0.1559	0.2659	0.232	0.516	0.1405	0.3496	0.8434
ExpectRepeat	0.3538	0.3591	0.3757	0.3428	0.3538	0.3591	0.3691	0.3428
Model	0.7863	0.3672	0.6021	0.7418	0.8121	0.2175	0.5845	0.7604
	$k = 15$				$k = 20$			
	Recall	**Precision**	**NDCG**	**HR**	**Recall**	**Precision**	**NDCG**	**HR**
RandomSet	0.0021	0.0006	0.0031	0.0002	0.0036	0.0007	0.0052	0.0000
TopKFrequent	0.5743	0.1128	0.374	0.9180	0.6214	0.101	0.3953	0.9180
ExpectRepeat	0.3538	0.3591	0.3683	0.3428	0.3538	0.3591	0.3682	0.3428
Model	0.8363	0.1564	0.5968	0.7890	0.8654	0.1223	0.6066	0.8294

Table B.5: Evaluation of different methods on the **IoT** data set with graph, $l = 30$

PhoneLabs	$N = 4729, l = 30, m = 2$, with graph							
	$k = 5$				$k = 10$			
Method	**Recall**	**Precision**	**NDCG**	**HR**	**Recall**	**Precision**	**NDCG**	**HR**
RandomSet	0.001	0.0006	0.0014	0.0000	0.0022	0.0007	0.0031	0.0002
TopKFrequent	0.8952	0.5371	0.8784	0.9022	0.9214	0.2764	0.891	0.9182
ExpectRepeat	0.9278	0.9281	0.9286	0.9104	0.9278	0.9281	0.9286	0.9104
Model	0.945	0.5568	0.9376	0.9328	0.9556	0.2827	0.9411	0.9428
	$k = 15$				$k = 20$			
	Recall	**Precision**	**NDCG**	**HR**	**Recall**	**Precision**	**NDCG**	**HR**
RandomSet	0.0031	0.0006	0.0043	0.0002	0.0042	0.0006	0.0059	0.0000
TopKFrequent	0.9396	0.1877	0.8977	0.9258	0.9454	0.1415	0.8995	0.9320
ExpectRepeat	0.9278	0.9281	0.9286	0.9104	0.9278	0.9281	0.9286	0.9104
Model	0.9619	0.1845	0.9427	0.9534	0.9652	0.1369	0.9445	0.9566

Table B.6: Evaluation of different methods on the **PhoneLabs** data set with graph, $l = 30$

IoT	$N = 5801, l = 40, m = 2$, with graph							
	$k = 5$				$k = 10$			
Method	**Recall**	**Precision**	**NDCG**	**HR**	**Recall**	**Precision**	**NDCG**	**HR**
RandomSet	0.0009	0.0006	0.0013	0.0000	0.0017	0.0007	0.0026	0.0000
TopKFrequent	0.3162	0.156	0.2679	0.2312	0.5232	0.1406	0.3530	0.8410
ExpectRepeat	0.3603	0.3651	0.3822	0.3472	0.3603	0.3651	0.3758	0.3472
Model	0.7632	0.3627	0.6075	0.6832	0.8158	0.2145	0.5894	0.7656
	$k = 15$				$k = 20$			
	Recall	**Precision**	**NDCG**	**HR**	**Recall**	**Precision**	**NDCG**	**HR**
RandomSet	0.0022	0.0006	0.0032	0.0002	0.0031	0.0006	0.0047	0.0002
TopKFrequent	0.5823	0.1126	0.3777	0.9226	0.6248	0.0993	0.3969	0.9226
ExpectRepeat	0.3603	0.3651	0.375	0.3472	0.3603	0.3651	0.375	0.3472
Model	0.8542	0.1559	0.6030	0.8082	0.8689	0.1205	0.6030	0.8460

Table B.7: Evaluation of different methods on the **IoT** data set with graph, $l = 40$

PhoneLabs	$N = 4729, l = 40, m = 2$, with graph							
	$k = 5$				$k = 10$			
Method	**Recall**	**Precision**	**NDCG**	**HR**	**Recall**	**Precision**	**NDCG**	**HR**
RandomSet	0.0013	0.0008	0.0018	0.0000	0.0023	0.0007	0.0032	0.0000
TopKFrequent	0.8957	0.5374	0.8774	0.906	0.9238	0.2771	0.8909	0.9236
ExpectRepeat	0.9324	0.9326	0.9332	0.9170	0.9324	0.9326	0.9331	0.9170
Model	0.9505	0.5581	0.9444	0.9392	0.9602	0.2842	0.9426	0.9512
	$k = 15$				$k = 20$			
	Recall	**Precision**	**NDCG**	**HR**	**Recall**	**Precision**	**NDCG**	**HR**
RandomSet	0.0034	0.0007	0.0047	0.0000	0.0039	0.0006	0.0054	0.0002
TopKFrequent	0.9447	0.1886	0.8985	0.9346	0.9506	0.1422	0.9004	0.9410
ExpectRepeat	0.9324	0.9326	0.9331	0.9170	0.9324	0.9326	0.9331	0.9170
Model	0.9639	0.1886	0.9474	0.9550	0.9666	0.1394	0.9457	0.9586

Table B.8: Evaluation of different methods on the **PhoneLabs** data set with graph, $l = 40$

B.3 Model Evaluation of Different Methods by use of the Dependency Graph Component

IoT	$N = 5801, l = 20, m = 2$, without graph							
	$k = 5$				$k = 10$			
Method	**Recall**	**Precision**	**NDCG**	**HR**	**Recall**	**Precision**	**NDCG**	**HR**
RandomSet	0.0009	0.0007	0.0015	0.0000	0.0016	0.0006	0.0024	0.0000
TopKFrequent	0.3129	0.1546	0.2655	0.2332	0.5152	0.1393	0.3489	**0.8430**
ExpectRepeat	0.3578	0.3624	0.3792	0.3444	0.3578	**0.3625**	0.3729	0.3444
Model	**0.6487**	**0.4472**	**0.6803**	**0.7168**	**0.7048**	0.2546	**0.6699**	0.7588
	$k = 15$				$k = 20$			
	Recall	**Precision**	**NDCG**	**HR**	**Recall**	**Precision**	**NDCG**	**HR**
RandomSet	0.0019	0.0006	0.0030	0.0006	0.0033	0.0007	0.0048	0.0000
TopKFrequent	0.5743	0.1119	0.3736	**0.9264**	0.6202	0.1000	0.3945	**0.9264**
ExpectRepeat	0.3577	**0.3624**	0.3720	0.3444	0.3578	**0.3624**	0.3720	0.3444
Model	**0.7215**	0.1735	**0.6226**	0.7734	**0.7393**	0.1344	**0.6190**	0.8048

Table B.9: Evaluation of different methods on the **IoT** data set without graph, $m = 2$

PhoneLabs	$N = 4729, l = 20, m = 2$, without graph							
	$k = 5$				$k = 10$			
Method	**Recall**	**Precision**	**NDCG**	**HR**	**Recall**	**Precision**	**NDCG**	**HR**
RandomSet	0.0008	0.0005	0.0011	0.0000	0.0019	0.0006	0.0027	0.0000
TopKFrequent	0.8954	0.5372	0.8803	0.8992	0.9198	0.2759	0.8919	0.9174
ExpectRepeat	0.9280	**0.9276**	0.9289	0.9102	0.9280	**0.9276**	0.9289	0.9102
Model	**0.9440**	0.5651	**0.9411**	**0.9348**	**0.9551**	0.2858	**0.9451**	**0.9476**
	$k = 15$				$k = 20$			
	Recall	**Precision**	**NDCG**	**HR**	**Recall**	**Precision**	**NDCG**	**HR**
RandomSet	0.0036	0.0007	0.0049	0.0000	0.0049	0.0007	0.0068	0.0002
TopKFrequent	0.9394	0.1876	0.8992	0.9286	0.9449	0.1414	0.9009	0.9346
ExpectRepeat	0.9280	**0.9276**	0.9289	0.9102	0.9281	**0.9276**	0.9289	0.9102
Model	**0.9601**	0.1916	**0.9452**	**0.9538**	**0.9646**	0.1444	**0.9481**	**0.9580**

Table B.10: Evaluation of different methods on the **PhoneLabs** data set without graph, $m = 2$

IoT	$N = 5801, l = 20, m = 2$, with graph							
	$k = 5$				$k = 10$			
Method	**Recall**	**Precision**	**NDCG**	**HR**	**Recall**	**Precision**	**NDCG**	**HR**
RandomSet	0.0006	0.0004	0.0009	0.0000	0.0018	0.0007	0.0025	0.0000
TopKFrequent	0.3122	0.153	0.2626	0.229	0.5188	0.1397	0.3482	0.849
ExpectRepeat	0.3633	0.3677	0.3846	0.3526	0.3633	0.3677	0.3781	0.3526
Model	0.7622	0.3598	0.6499	0.6592	0.7587	0.2054	0.5938	0.6098
	$k = 15$				$k = 20$			
	Recall	**Precision**	**NDCG**	**HR**	**Recall**	**Precision**	**NDCG**	**HR**
RandomSet	0.0026	0.0007	0.0037	0.0000	0.0029	0.0006	0.0043	0.0004
TopKFrequent	0.5756	0.1117	0.372	0.9256	0.6222	0.1001	0.3931	0.9256
ExpectRepeat	0.3633	0.3677	0.3773	0.3526	0.3633	0.3677	0.3773	0.3526
Model	0.8078	0.1498	0.5890	0.6908	0.8164	0.1183	0.5869	0.7074

Table B.11: Evaluation of different methods on the **IoT** data set with graph incorporated in the model

PhoneLabs	$N = 4729, l = 20, m = 2$, with graph							
	$k = 5$				$k = 10$			
Method	**Recall**	**Precision**	**NDCG**	**HR**	**Recall**	**Precision**	**NDCG**	**HR**
RandomSet	0.0013	0.0008	0.0018	0.0000	0.002	0.0006	0.0029	0.0002
TopKFrequent	0.9022	0.5413	0.8871	0.907	0.9252	0.2775	0.8981	0.9208
ExpectRepeat	0.9310	0.9310	0.9317	0.9128	0.931	0.931	0.9316	0.9128
Model	0.9503	0.5236	0.9437	0.9376	0.9592	0.2814	0.9473	0.9506
	$k = 15$				$k = 20$			
	Recall	**Precision**	**NDCG**	**HR**	**Recall**	**Precision**	**NDCG**	**HR**
RandomSet	0.0033	0.0007	0.0046	0.0002	0.0045	0.0007	0.0062	0.0002
TopKFrequent	0.943	0.1883	0.9046	0.9326	0.9498	0.1421	0.9067	0.9400
ExpectRepeat	0.931	0.9310	0.9316	0.9128	0.9310	0.9310	0.9316	0.9128
Model	0.9638	0.1900	0.9499	0.9574	0.9672	0.1427	0.9510	0.9638

Table B.12: Evaluation of different methods on the **PhoneLabs** data set with graph incorporated in the model

B.4 Model Evaluation of Different Methods by use of the Repeated Element Component

IoT	$N = 5801, l = 20, m = 2$, with graph							
	$k = 5$				$k = 10$			
Method	**Recall**	**Precision**	**NDCG**	**HR**	**Recall**	**Precision**	**NDCG**	**HR**
RandomSet	0.0008	0.0006	0.0013	0.0000	0.0018	0.0007	0.0026	0.0000
TopKFrequent	0.3105	0.1537	0.2639	0.2238	0.5184	0.1401	0.35	0.8466
ExpectRepeat	0.3634	0.3675	0.385	0.3494	0.3634	0.3675	0.3788	0.3494
Model	0.7817	0.4176	0.6673	0.7048	0.8040	0.2318	0.6487	0.7034
	$k = 15$				$k = 20$			
	Recall	**Precision**	**NDCG**	**HR**	**Recall**	**Precision**	**NDCG**	**HR**
RandomSet	0.0024	0.0007	0.0036	0.0004	0.0028	0.0006	0.004	0.0002
TopKFrequent	0.5769	0.1123	0.373	0.9256	0.6224	0.1002	0.3949	0.9278
ExpectRepeat	0.3536	0.357	0.3674	0.3392	0.3617	0.3673	0.3759	0.3464
Model	0.8364	0.1707	0.6402	0.7662	0.8742	0.1322	0.6426	0.8228

Table B.13: Evaluation of different methods on the **IoT** data set with graph, without repeated element component γ

PhoneLabs	$N = 4729, l = 20, m = 2$, with graph							
	$k = 5$				$k = 10$			
Method	**Recall**	**Precision**	**NDCG**	**HR**	**Recall**	**Precision**	**NDCG**	**HR**
RandomSet	0.0014	0.0008	0.0019	0.0000	0.0015	0.0004	0.0021	0.0002
TopKFrequent	0.8985	0.5391	0.8826	0.9024	0.9235	0.277	0.8945	0.9186
ExpectRepeat	0.9326	0.9328	0.9332	0.9146	0.9326	0.9328	0.9332	0.9146
Model	0.9431	0.5580	0.9411	0.9252	0.9493	0.2606	0.9424	0.9362
	$k = 15$				$k = 20$			
	Recall	**Precision**	**NDCG**	**HR**	**Recall**	**Precision**	**NDCG**	**HR**
RandomSet	0.0031	0.0006	0.0044	0.0000	0.0042	0.0006	0.0057	0.0006
TopKFrequent	0.9425	0.1882	0.9015	0.9288	0.9493	0.142	0.9036	0.939
ExpectRepeat	0.9326	0.9328	0.9332	0.9146	0.9326	0.9328	0.9332	0.9146
Model	0.9532	0.1713	0.9428	0.9390	0.9549	0.1386	0.9436	0.9424

Table B.14: Evaluation of different methods on the **PhoneLabs** data set with graph, without repeated element component γ

B.5 Model Evaluation of Different Methods by Partial QEP Size

IoT	$N = 5801, l = 20, m = 2$, with graph									
	Recall, $k = 5$									
Method	1	2	3	4	5	6	7	8	9	10
RandomSet	0.0011	0.0006	0.0004	0.0005	0.0000	0.0000	0.0001	0.0000	0.0000	0.0000
TopKFrequent	0.3823	0.3103	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
ExpectRepeat	0.4040	0.3471	0.1359	0.0905	0.0040	0.0038	0.0019	0.0000	0.0000	0.0000
Model	0.8024	0.7250	0.3274	0.2293	0.0878	0.1126	0.0390	0.0042	0.0170	0.0132
	Recall, $k = 10$									
Method	1	2	3	4	5	6	7	8	9	10
RandomSet	0.0019	0.0014	0.0009	0.0006	0.0002	0.0000	0.0002	0.0000	0.0000	0.0000
TopKFrequent	0.8350	0.4050	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
ExpectRepeat	0.4040	0.3471	0.1359	0.0905	0.0040	0.0038	0.0019	0.0000	0.0000	0.0000
Model	0.8857	0.8003	0.3462	0.2355	0.0999	0.1247	0.0610	0.0047	0.0180	0.0108
	Recall, $k = 15$									
Method	1	2	3	4	5	6	7	8	9	10
RandomSet	0.0031	0.0028	0.0017	0.0004	0.0004	0.0000	0.0003	0.0000	0.0000	0.0000
TopKFrequent	0.9499	0.4383	0.0522	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
ExpectRepeat	0.4040	0.3471	0.1359	0.0905	0.0040	0.0038	0.0019	0.0000	0.0000	0.0000
Model	0.9217	0.8395	0.3586	0.2474	0.1201	0.1479	0.0707	0.0051	0.0120	0.0109
	Recall, $k = 20$									
Method	1	2	3	4	5	6	7	8	9	10
RandomSet	0.0030	0.0039	0.0021	0.0015	0.0006	0.0006	0.0004	0.0001	0.0000	0.0000
TopKFrequent	0.9499	0.5035	0.1174	0.0652	0.0000	0.0652	0.0000	0.0000	0.0000	0.0000
ExpectRepeat	0.4040	0.3471	0.1359	0.0905	0.0040	0.0038	0.0019	0.0000	0.0000	0.0000
Model	0.9389	0.8528	0.3688	0.2528	0.1224	0.1475	0.0714	0.0051	0.0117	0.0109

Table B.15: Recall of different methods on the **IoT** data set by Partial QEP size

PhoneLabs	$N = 4729, l = 20, m = 2$, with graph									
	Recall, $k = 5$									
Method	1	2	3	4	5	6	7	8	9	10
RandomSet	0.0013	0.0014	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
TopKFrequent	0.8852	0.9143	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
ExpectRepeat	0.9162	0.9184	0.0130	0.0069	0.0028	0.0020	0.0018	0.0008	0.0000	0.0000
Model	0.9346	0.9363	0.0229	0.0082	0.0016	0.0026	0.0009	0.0003	0.0000	0.0004
	Recall, $k = 10$									
Method	1	2	3	4	5	6	7	8	9	10
RandomSet	0.0019	0.0021	0.0002	0.0001	0.0002	0.0001	0.0000	0.0000	0.0000	0.0000
TopKFrequent	0.9303	0.9143	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
ExpectRepeat	0.9162	0.9184	0.0130	0.0069	0.0028	0.0020	0.0018	0.0008	0.0000	0.0000
Model	0.9400	0.9418	0.0248	0.0114	0.0033	0.0026	0.0011	0.0003	0.0000	0.0004
	Recall, $k = 15$									
Method	1	2	3	4	5	6	7	8	9	10
RandomSet	0.0034	0.0041	0.0001	0.0001	0.0000	0.0000	0.0000	0.0000	0.0001	0.0001
TopKFrequent	0.9377	0.9303	0.0160	0.0042	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
ExpectRepeat	0.9162	0.9184	0.0130	0.0069	0.0028	0.0020	0.0018	0.0008	0.0000	0.0000
Model	0.9405	0.9437	0.0270	0.0139	0.0040	0.0027	0.0018	0.0003	0.0001	0.0005
	Recall, $k = 20$									
Method	1	2	3	4	5	6	7	8	9	10
RandomSet	0.0041	0.0051	0.0003	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
TopKFrequent	0.9377	0.9349	0.0194	0.0042	0.0042	0.0013	0.0000	0.0000	0.0000	0.0000
ExpectRepeat	0.9162	0.9184	0.0130	0.0069	0.0028	0.0020	0.0018	0.0008	0.0000	0.0000
Model	0.9414	0.9465	0.0276	0.0161	0.0066	0.0031	0.0022	0.0003	0.0001	0.0005

Table B.16: Recall of different methods on the **PhoneLabs** data set by Partial QEP size