TU WIEN Informatics

Umschreiben oder nicht umschreiben: Entscheidungsfindung bei der Anfrageoptimierung von SQL Anfragen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Data Science

eingereicht von

Daniela Böhm, BSc.
Matrikelnummer 11918462

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Mag.rer.nat. Dr.techn. Reinhard Pichler
Mitwirkung: Univ.Ass. Dipl.-Ing. Alexander Selzer
Assistant Prof. Dipl.-Ing. Dr.techn. Matthias Lanzinger

Wien, 21. August 2024

Daniela Böhm	Reinhard Pichler

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

To rewrite or not to rewrite: Decision making in query optimization of SQL queries

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Data Science

by

Daniela Böhm, BSc.
Registration Number 11918462

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Mag.rer.nat. Dr.techn. Reinhard Pichler
Assistance: Univ.Ass. Dipl.-Ing. Alexander Selzer
Assistant Prof. Dipl.-Ing. Dr.techn. Matthias Lanzinger

Vienna, August 21, 2024

Daniela Böhm | Reinhard Pichler

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Daniela Böhm, BSc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 21. August 2024

Daniela Böhm

Danksagung

An erster Stelle möchte ich mich bei meinem Betreuer Prof. Dr. Reinhard Pichler für den Themenvorschlag sowie die zeitliche und inhaltliche Unterstützung, die weit über das übliche Maß hinausgegangen ist, ganz herzlich bedanken. Mein Dank gilt weiters Dipl.-Ing. Alexander Selzer für das stets prompte und kompetente Antworten auf meine Fragen zum Programmierteil und Prof. Dr. Matthias Lanzinger für die hilfreiche Unterstützung beim Machine Learning Teil. Es ist etwas Besonderes, so intensiv von einem Team bei der Masterarbeit unterstützt zu werden.

Die Arbeit wurde im Rahmen des Forschungsprojekts, das vom Wiener Wissenschafts-, Forschungs- und Technologiefonds (WWTF) [10.47379/ICT2201] gefördert wird, durchgeführt. Weiters wurde diese Arbeit durch die dataLAB/Big Data Infrastruktur der TU Wien ermöglicht. Ich danke dem TU.it dataLAB Big Data-Team der TU Wien für die Unterstützung.

Ein besonderer Dank gebührt meinen Eltern, die immer hinter mir stehen und mich darin bestärken, meinen Weg zu gehen. Ohne ihre kontinuierliche Unterstützung wäre ich heute nicht da, wo ich bin. Außerdem möchte ich mich bei meiner Familie und meinem Umfeld explizit dafür bedanken, dass sie mich von klein auf ermutigt haben, auch als Mädchen und Frau ein Interesse an mathematischen und technischen Bereichen zu entwickeln und mir mein Studium zugetraut haben.

Unglaublich dankbar bin ich für die beste Lerngruppe, Paul Czapka und Hannes Mayrhofer, ohne die das Studium nicht in dieser Form möglich gewesen wäre. Die gemeinsame Zeit, die Freundschaft und das gegenseitige Unterstützen sind unbezahlbar. Meine Partnerschaft mit Pauli ist das Allerschönste, das aus dieser Lerngruppe entstehen konnte, und dafür bin ich unendlich dankbar.

Acknowledgements

First of all, I would like to thank my supervisor Prof. Dr. Reinhard Pichler for the thesis' topic suggestion as well as the support in terms of time and content, which went far beyond the usual. Additionally, I would like to thank Dipl.-Ing. Alexander Selzer for his fast and competent answers to my questions about the programming part and Prof. Dr. Matthias Lanzinger for his support concerning the Machine Learning part. It is truly special to receive such intensive support throughout the work on the master thesis.

The work on this thesis was carried out in the context of the research project funded by the Vienna Science and Technology Fund (WWTF) [10.47379/ICT2201]. The production of this work has been enabled by the dataLAB/Big Data infrastructure @TU-Wien. I acknowledge the assistance of the TU.it dataLAB Big Data team at TU-Wien.

Special thanks to my parents for always having my back and for encouraging me to follow my path. Without their continous support I would not be, where I am today. Furthermore, I would like to thank my family and those around me, who always encouraged me to follow my interests in mathematical and techinal fields, even as a girl and as a woman, and who believed in me and my ability to finish my studies.

I am incredibly thankful for the best study group, Paul Czapka and Hannes Mayrhofer, without whom it would not have been possible to study in the way we did. The time spent together, the friendship and the support of each other are priceless. The romantic relationship with Pauli is the most beautiful thing that could have come from that study group and I am infinetely grateful for that.

Kurzfassung

Eine typische Herausforderung für Datenbankmanagementsysteme (DBMSs) ist es, Queries effizient auszuwerten. Die einfachsten Queries sind Conjunctive Queries (CQs), die in SQL SELECT-FROM-WHERE Queries entsprechen, bei denen im WHERE statement nur Gleichheitsbedingungen und logische Unds (AND) erlaubt sind. Sogar das Auswerten dieser fundamentalen Queries ist ein NP-vollständiges Problem.

In der Praxis ist ein erheblicher Teil aller Queries azyklisch oder fast azyklisch, die CQs mit einfacheren Strukturen sind. DBMSs berücksichtigen strukturelle Eigenschaften im Normalfall nicht, wohingegen in der Theorie mit dem Yannakakis Algorithmus eine effiziente Auswertungsmethode für azyklische Queries existiert. Um eine auf Yannakakis basierende Auswertungsmethode zu nutzen, muss die Query umgeschrieben werden, sodass das DBMS gezwungen wird, die Query in der Art auszuführen, die Yannakakis vorschlägt. Es gibt einen Ansatz, der solch eine Umschreibungsmethode, die on-top von einigen DBMSs benutzt werden kann, für azyklische CQs mit zusätzlichen Aggregaten bereitstellt. Theoretisch wird der asympotitische Worst-Case immer besser, wenn man diese Methode benutzt. Allerdings werden in der Praxis zusätzliche Overheads produziert und es ist unklar und schwierig zu entscheiden, ob die Umschreibungsmethode oder das Auswerten mit dem ursprünglichen DBMS vorteilhafter ist.

Daher wird ein Entscheidungsprogramm benötigt, um herauszufinden, ob es besser ist, die Query umzuschreiben oder in ihrer originalen Form zu verwenden. Die Aufgabe dieser Arbeit ist es, solch ein Entscheidungsprogramm zu entwickeln und zu implementieren. Das wird mit Hilfe von umfangreichen Tests auf Benchmarkdatensätzen gemacht, um Features zu finden, mit denen man die Queries unterscheiden kann. Auf Basis dieser Features wird das Entscheidungsprogramm entwickelt und programmiert. Das Entscheidungsprogramm ist ein Machine Learning Modell, das aus einigen modernen Machine Learning Modellen ausgewählt wird.

Bei unseren quantitativen und qualitativen Analysen zeigt sich, dass der Decision Tree am besten funktioniert. Dafür werden Metriken benutzt, die fehlklassifizierte Fälle untersuchen und statistische Tests herangezogen. Weiters sind Decision Trees Modelle, die interpretiert werden können und die keinen hohen Rechenaufwand erfordern. Mit diesem Decision Tree als Entscheidungsprogramm können wir drei komplett unterschiedliche DBMSs, nämlich PostgreSQL, DuckDB and SparkSQL, übertreffen.

Abstract

A common challenge for database management systems (DBMSs) is efficiently evaluating queries. The most basic queries are conjunctive queries (CQs), which are SELECT-FROM-WHERE queries only allowing equality conditions with logical ands (AND) in the WHERE statement in SQL. Even the evaluation of these fundamental queries is an NP-complete problem.

In practice a significant portion of queries is acyclic or almost acyclic, which are CQs with easier structures. DBMSs generally do not consider structural properties, but in theory Yannakakis' algorithm gives us an efficient evaluation for acyclic queries. To make use of Yannakakis-style evaluation the query has to be rewritten such that the DBMS is forced to execute the query like Yannakakis' algorithm would suggest. There is an approach providing such a rewriting method applicable on-top of several DBMSs for acyclic CQs allowing additional aggregates. In theory, the asymptotic worst case always gets better using this method. Nevertheless, in practice additional overheads are produced and it is unclear and hard to decide, whether it is preferable to use the rewriting method or the plain DBMS for the evaluation.

Therefore, a decision program is needed to determine, if the query should be rewritten or evaluated in its original form. The purpose of this work is to design and implement such a program. This is done by using extensive testing on benchmark datasets to find out which features can be used to distinguish the queries. Based on these features the decision method is designed and implemented. The decision program is a Machine Learning model chosen out of a range of modern Machine Learning models.

We see that the decision tree performs best in terms of quantitative and qualitative analysis, based on metrics, inspection of misclassifications and statistical tests. Moreover, decision trees are interpretable models, which are computationally not expensive. With this decision tree as decision program we can outperform three completely different existing DBMSs, namely PostgreSQL, DuckDB and SparkSQL.

Contents

CHAPTER 1

Introduction

Background Over the past few years the amount of available data has been and is still increasing enormously. Consequently, there is a significant amount of research in data-driven areas aimed at efficiently storing and working with data. Database management systems (DBMSs) play an important role, since they give us the opportunity to access data stored in databases using queries. The most basic queries are conjunctive queries (CQs). In SQL they are SELECT-FROM-WHERE queries only allowing equality conditions with logical ands (AND) in the WHERE statement. In the context of relational algebra, they are SELECT-PROJECT-JOIN statements with equality conditions.

Even the evaluation of these fundamental queries is an NP-complete problem, which was shown by Chandra and Merlin, 1977. Because of that the evaluation of big join queries can lead to an explosion of the intermediate results and to very long execution times. Classical DBMSs try to use heuristics to find a join order, which reduce to huge intermediate results, but they cannot eliminate the problem.

A reason for that is, that DBMSs rarely use the query structure to decide which execution plan would be the best. In theory, there are results, which lead to efficient evaluation times of acyclic join queries based on Yannakakis' algorithm, introduced in Yannakakis, 1981, using the query structure. Even if acyclicity looks like a severe restriction, in practice a significant portion of the queries is alcyclic or almost acyclic as shown by Bonifati et al., 2017 and Fischl et al., 2021. Yannakakis-style query evaluation is not yet implemented in DBMSs.

Problem Statement Since it is not an easy process to change one or multiple DMBSs in a way that they use the theoretical results for evaluation, another possibility is an on-top-rewriting procedure. This means that the query is rewritten in a way, that it forces the DBMS to execute the query like Yannakakis' algorithm would suggest. That approach is not only easier to implement, it also gives us the possibility to use the rewriting on top of multiple DBMSs with only very small changes necessary.

Such an on-top-rewriting has recently been implemented in Gottlob et al., 2023. It uses a Yannakakis-based approach. This means, it evaluates the query in three stages, the bottom-up, the top-down and a second bottom-up traversal of the join tree. During the first two traversals only semi-joins are used and all dangling tuples, which means tuples, that do not contribute to the final result, are eliminated. In the end, joins are performed to get the final result, but these joins only produce tuples, which are part of the final result. This approach automatically leads to a good join order and prevents any unnecessary intermediate results. In theory, the used approach is always faster (or the same) than the plain evaluation for the asymptotic worst case. Unfortunately, in practice it can be worse, because an overhead is produced. This means the semi-join results need to be saved in auxiliary tables and the three traversals can lead to additional workload, which sometimes does not pay off, e.g. when no or only a few dangling tuples exist. The method proposed in Gottlob et al., 2023, which is our basis, significantly reduces the evaluation time for about half of the queries, but for the other half the plain DBMS is faster. Therefore, neither version should be preferred in general. Unfortunately, there is no clear pattern to identify which version will be the faster one. Therefore, there is the need for a program deciding if the query should be rewritten or evaluated in its original form.

Goal The purpose of this thesis is to design such a program, which makes the right decision in as many cases as possible. The decision program is a Machine Learning model. One goal is to find out which model type performs best and should be the final decision program. We use the following modern Machine Learning models as candidates: k-nearest neighbors, decision trees, random forests, support vector machines, multi-layer perceptrons, hypergraph neural networks and a combination of the latter two. Additionally, the hyperparameters of each model are varied and compared to achieve better results. The features for the Machine Learning models are structural properties and/or data properties of the queries. It is another goal to find out, which features are suitable for our task and help making a good decision. The third big aim is then to observe, if it is possible to outperform existing DBMSs. Outperforming means to be in the same or a smaller order of magnitude regarding the evaluation time. The evaluation of the decision program and the execution of the query with the chosen version should be faster than the evaluation of the query on the plain DBMS. For cases where rewriting gains performance, we want to decide to rewrite, perform rewriting and evaluate faster than the DBMS. For cases where we decide not to rewrite, we will be slower, since we add the time of deciding and then executing in the same way as the DBMS does anyway, but we want to keep this additional effort as small as possible.

Approach To be able to achieve goals mentioned above, we perform extensive testing. On the one hand, we use three completely different DBMSs, namely PostgreSQL, DuckDB and SparkSQL. PostgreSQL is a well-established relational DBMS, whereas DuckDB is an in-process, column-oriented DBMS, and SparkSQL is part of a distributed cluster environment. Therefore, these three give a good coverage of the existing DBMSs. On the other hand, we try to find as many suitable benchmark datasets as possible and

evaluate the queries of them once in their original form and once as rewritten version. These runtimes are the responses for our Machine Learning models (either as labels, if the original or rewritten version is faster as classification, or as time differences for regression). Additionally, we use different features from the literature, e.g. Abseher et al., 2017, and design some on our own, which then are the input features for our Machine Learning models.

As benchmark datasets we use the STATS dataset introduced by Han et al., 2021, the SNAP (Stanford Network Analysis Project) dataset introduced by Leskovec and Krevl, 2014, the JOB (Join Order Benchmark) dataset introduced by Leis et al., 2015, the LSQB (Large-Scale Subgraph Query Benchmark) dataset introduced by Mhedhbi et al., 2021 and the HETIONET dataset introduced by Himmelstein et al., 2017. They all have different purposes like testing the join order or query optimizers, using different joins and foreign key relations and representing graph datasets of real world data. For our purpose it is important to have these different datasets to represent a broad range of use cases.

Results Using this approach with the mentioned DBMSs and benchmark datasets, we achieve our main goal, to correctly decide which version is the faster one, in most cases. The final model is the decision tree and achieves high accuracy scores of at least 82% and up to about 94% and high precision values (up to 96%). Moreover, we are able to outperform the existing DBMSs as we show with statistical tests, the mean and median runtimes are significantly smaller for the decision program than for the plain DBMS. This is the case for all our settings for all DBMSs. Additionally, most of the misclassifications are cases, where the original and rewritten runtimes are very similar anyway, which means the negative effect of the misclassification is negligable. Moreover, decision trees are interpretable and computationally not expensive. Furthermore, we use the structure of the decision tree to find out, which features are most important.

Structure of the thesis In Chapter 2 we start by providing related work to our topic. Then, the theoretical background of CQs and their characteristics are given in Chapter 3, followed by explaining hypergraphs and acyclicity in Chapter 4. This includes providing the GYO-reduction for recognizing acyclicity and Yannakakis' algorithm, which is the basis for fast evaluation of acyclic queries. Based on that, Chapter 5 recalls several query decomposition methods to give an overview of how more complex queries can be handled, including the 0MA query class, which has recently been introduced in Gottlob et al., 2023. Afterwards, we give a short introduction to Machine Learning in Chapter 6, where we describe all used models and the experiment design of model selection, which we perform to choose the "best" model as our decision program. In Chapter 7 the methodology and workflow of the practical part of the thesis are described, followed by the obtained results in Chapter 8. In the end the results, limitations and future work are summarized in Chapter 9.

CHAPTER 2

Related Work

Yannakakis' algorithm in Database Theory
To obtain several complexity results of query enumeration the following papers are given, where Yannakakis' algorithm is one substantial part for obtaining each of these results. Bagan et al., 2007 discuss the enumeration complexity of ACQs with additional disequality conditions. They show that there is a big subclass of CQs (self-join-free ACQs) which can be efficiently enumerated with constant delay. This means (after a linear time precomputation) each enumeration result of the query can be generated after the last one in constant time. This is a good computational result.

Additional results of enumeration complexity are given in Carmeli and Kröll, 2020. For CQs with functional dependencies they show that the enumeration can be done with constant delay and following linear-time preprocessing, which was previously considered not true. If there are no such dependencies the enumreration does not have such favorable behaviour. Afterwards they showed that unions of CQs can also be enumarted with constant delay and following linear-time preprocessing, even if some or sometimes all single CQs of them are does not have this characteristic in Carmeli and Kröll, 2021. In Carmeli et al., 2021 algorithms for efficient access to ordered query results using direct access or selection are discussed for conjunctive queries without self-joins. They also find some conditions when the algorithms are applicable and study the influence of functional dependencies on this problem. In Carmeli et al., 2022 the authors further inspected the behaviour using random access and random-order enumeration. They show that free-connex ACQs (i.e. query body is acyclic and is still acyclic, when head is added as additional atom) can be executed with constant delay and following linear-time prepocessin for enumeration, random enumeration and random access, whereas all other queries cannot. Geck et al., 2022 show if for an ACQ a rewriting with an equivalent result exists, then there exists at least one rewriting, which is acyclic, too.

Yannakakis-style evaluation in Database Systems
After discussing some theoretical results, we want to present works using Yannakakis'

algorithm for practical purposes. One of the first Yannakakis-style methods is introduced in Ghionna et al., 2007. They extend the hypertree decomposition notion for query optimization and implement an optimizer for PostgreSQL using structure-guided query processing.

In Tu and Ré, 2015 Yannakakis' algorithm and a second (worst-case optimal) algorithm introduced by Ngo et al., 2012 are compared and used to explore query planning. The join order of queries is often not optimal and in this paper generalized hypertreee decompositions are used together with the two algorithms to achieve better results. Based on this, Perelman and Ré, 2015 discuss a query compiler (DunceCape), which outperforms traditional RDBMS algorithms. Upon this EmptyHeaded, a new high-level join engine architecture is built by Aberger et al., 2017, which outperforms several existing systems.

For free-connex ACQs a dynamic version of Yannakakis' algorithm was designed in Idris et al., 2017. It allows constant-delay enumeration and constant-time lookups, handles updates efficiently and the amount of memory and storage is linearly proportional to the size of the database for these queries. In addition, the authors prove a general dynamic version of Yannakakis' algorithm, which also allows theta-joins in the queries in Idris et al., 2020 and provide a new algorithm for computing query plans in this framework. Q. Wang et al., 2023 use the idea of Yannakakis to replace joins with semi-joins to avoid big intermediate results, which are saved as materialized views in classical change propagation frameworks. The new framework they create still achieves constant-delay enumeration and outperforms existing systems. Predicates of a query, which are comparisons between at least two tables, cannot be pushed down efficiently in many cases. Therefore, in Q. Wang and Yi, 2022 a new algorithm is provided for evaluating such queries. Additionally, they found a class of queries, where this can be done in linear time.

A further interesting topic in practice is data security and processing of private data. To ensure privacy a secure two-party computation model can be used, where the data does not need to be revealed while evaluating a query. Y. Wang and Yi, 2021 propose a secure version of Yannakakis' algorithm to meet the privacy requirement, but also improve the evaluation time enormously compared to the state-of-the art model before.

In contrast to the algorithms and standalone models explained until now, Hu and Wang, 2023 are to the best of our knowledge the only ones, who are using a model on top of existing DBMSs. Despite this fact, their approach is different to ours. They are studying the set-difference of query results of several (small) queries.

Machine Learning for database system applications
In general, Machine Learning (AI/Deep Learning) and database systems can help improving each other. W. Wang et al., 2016 and Zhou et al., 2022 list applications, where Deep Learning can be used to solve or improve database problems such as cardinality estimation, knob tuning, join order selection or index creation. On the other hand, databases can help to access AI more easily, improve the training process and allow better parallelization. We want to focus on how Machine Learning can be used to improve some

aspects of databases, especially the join order selection and the selectivity estimation.

For query evaluation the join order, which is chosen by the database systems often based on heuristics, is crucial. Nevertheless, in pratice it is not always optimal and there are different approaches to address this issue. We want to present some papers using Reinforcement Learning or Deep Learning models to achieve a better join order. Marcus and Papaemmanouil, 2018 state the idea of using feedback of how well the chosen join order worked out for some queries, to learn from mistakes and adjust to them using a Reinforcement Learning agent. In Marcus et al., 2019 a new query optimizer based on deep learning models is introduced. To design the optimization model, bootstrapping of the optimization models of existing optimizers is used and upon this each new query is used to help the optimizer to learn how good the chosen join order was. A different approch of designing an optimizer is given by X. Yu et al., 2020. The current query is split into several subqueries and using Reinforcement Learning the optimal join order of the whole query is created. Another optimizer based on Reinforcement Learning with a tree-structured long short-term memory (LSTM) model is provided by Trummer et al., 2021. It takes the join tree into consideration for deciding the optimal join order and allows to change the database schema when this is beneficial.

The basis of choosing the join order and estimating the cost is the cardinality estimation, which also can be done by using Machine Learning techniques. In Kipf et al., 2018 a deep learning approach, namely a convolutional neural network, is provided to predict correlation between join tables, which can be used for the selectivity estimation. Dutt et al., 2019 use neural networks, too, but also compare them to tree-based ensembles and decide to treat the problem as regression problem. They additionally adjust the features as well as transform the regression response to outperform existing methods. Hasan et al., 2020 again provide a supervised lightweight neural network, but also consider another simple approach, which is using density estimation to estimate the joint probability distribution to gain insights into the selectivity.

In contrast to the three provided approaches before, Hilprecht et al., 2020 do not want to use pairs of queries and selectivities for training a model, but they want to introduce a data-driven approach. This means the database schema and database components are used for training a deep probabilistic model, which then is also able to predict the selectivity, but based on the data and not on the given queries. This approach allows to adjust to changes of the DBMS immediately and no queries are needed for training, but like for all data-driven approaches a lot of data is needed to make this model work. Wu and Cong, 2021 then combine the data-driven and the query-driven approaches and design a deep autoregressive model using both datatypes to estimate the joint probability, which again can be used for selectivity estimation.

Further research using Machine Learning in the database context is for example done by deciding, which indexes to use (e.g. Ding et al., 2020, Nathan et al., 2020, Kossmann et al., 2020) and by learning the database configurations (e.g.Van Aken et al., 2017, Zhang et al., 2019, Kunjir and Babu, 2020).

CHAPTER 3

Conjunctive queries

Database management systems (DBMSs) are used to interact with databases. Databases store data in a structured way as tables with rows and columns. With DBMSs a user can create, adjust, control and access the data in the database(s). A widely used approach is to access the databases with queries. Multiple actions like selecting columns, joining tables, filtering, grouping or counting can be performed by writing queries in a DBMS. In Section 3.1 we take a closer look at conjunctive queries, a class of simple queries. We provide different notations for those queries, namely in relational algebra notation, datalog notation and as SQL statements. Additionally, we line out that conjunctive queries are not only useful in the context of DBMSs, but also an important field of Artificial Intelligence and Operations Research applications. Finally, we talk about the complexity of the evaluation of conjunctive queries, which is NP-complete in Section 3.2.

3.1 Conjunctive queries (CQs)

The class of conjunctive queries (CQs) is arguably the most fundamental type of queries (for a definition see Chandra and Merlin, 1977). These are queries only allowing projections, equi-joins and selections with equality conditions combined by logical "AND"s. In **relational algebra** notation CQs have the form

$$\pi(\sigma(R_1 \bowtie ... \bowtie R_n))$$

where π represents the projections, σ the selections and $\bowtie$ the equi-joins. For each of these operations, for real world examples, the columns or conditions corresponding to the projection, selection or join are provided next to the symbol as subscripts.

To give some examples throughout this thesis based on a database, we define a database schema. We consider a database consisting of data about a university. There is a table `student` with the name, the immatriculation number and the study program of every enrolled student. Then, there is a table `masterthesis`, where the immatriculation number of all students writing a master thesis is listed together with the title of their thesis and a unique identification code of the professor, who is their advisor. The third table `professor` contains the name and the identification code of all professors. Additionally, a table `room` lists the room numbers of every office in the university, together with the identification code of the professor sitting there. In the table `program` all study programs are saved with the study program label and the faculty they belong to.

Our example database schema looks like the following:
`student`(`name`, `immatriculation_number`, `study_program`)
`masterthesis`(`title`, `immatriculation_number`, `advisor_code`)
`professor`(`name`, `identification_code`)
`room`(`number`, `identification_code`)
`program`(`name`, `label`, `faculty`)

In this chapter we are only using the relations `student` and `professor`. To simplify the statements, we only use the first letter for each relation and column, which means S[N,I,S] and P[P,C].

A CQ in relational algebra notation could now look like this

$$\pi_{P.P}(\sigma_{S.N=P.P}(S \bowtie_{S.I=P.C} P))$$

giving us the names of all professors, who are also students.

After defining CQs formally and with an example, we look at some properties they fulfill. Since only equality conditions are allowed, we could also write $\pi(R_1 \bowtie ... \bowtie R_n)$ without loss of generality. We are shortly explaining the two possible cases, either having two relations or one relation in the equality condition, and why we can rewrite it without selections in both cases. If a condition includes two relations in one equality condition, it is equivalent to an equi-join and we can add the condition to the equi-joins and do not need the selection anymore. As an example, we take the same CQ as before and rewrite it without a selection.

$$\pi_{P.P}(\sigma_{S.N=P.P}(S \bowtie_{S.I=P.C} P)) \Leftrightarrow \pi_{P.P}(S \bowtie_{S.I=P.C \wedge S.N=P.P} P))$$

In the second case, if a condition only includes columns of one relation, it is a filter and we can apply it separately on the relation and only use the filtered relations for the joins.

Here is an example with a selection only using the relation S in the selection

$$\pi_{P.P}(\sigma_{S.I=11918450}(S \bowtie_{S.I=P.C} P)) \Leftrightarrow \pi_{P.P}((\sigma_{S.I=11918450}S) \bowtie_{S.I=P.C} P)) \\ \Leftrightarrow \pi_{P.P}(\tilde{S} \bowtie_{\tilde{S}.I=P.C} P)$$

where $\tilde{S}$ contains only the tuples of relation S, where $S.I = 11918450$.

Another property which we are going to use is that the equi-joins with the equality conditions are equivalent to natural joins after renaming some columns accordingly. Using the same relations again, one example is

$$\pi_{P.P}(S \bowtie_{S.I=P.C} P) \Leftrightarrow \pi_{P.P}((\rho_{C \leftarrow I}S) \bowtie P)$$

where $\rho_{C \leftarrow I}$ represents the renaming of the attribute "immatriculation_number" to "identification_code". So, after this operation `student` has the form S[N,C,S] instead of S[N,I,S].

After looking at some properties of CQs, we provide different notations for queries. In contrast to the relational algebra, which is an operational notation, we want to mention **SQL** as declarative query language. Very often SQL is used to interact with databases. Relational algebra expressions can be expressed with an equivalent SQL statement. CQs in SQL are SELECT-FROM-WHERE statements allowing only equality constraints together with ANDs. As an example $\pi_{P.P}(S \bowtie_{S.I=P.C} P)$ corresponds to

```
SELECT P.name
FROM student as S, professor as P
WHERE S.immatriculation_number = P.identification_code;
```

using the full name of the relations and attributes here.

Finally, we want to present **datalog notation** as a powerful notation combining operational and declarative concepts. It is very common to write CQs in **datalog notation**, which looks like this

$$Q(\vec{x}) : -R_1(\vec{z_1}), ..., R_n(\vec{z_n})$$

where $Q(\vec{x})$ is called the head of the CQ and $R_1(\vec{z_1}), ..., R_n(\vec{z_n})$ the body with atoms $R_i(\vec{z_i})$. Atom is a more commonly used word for relation in this context.

The same example as above can be written as

$$Q(P) : -S(N, C, S), P(P, C)$$

in datalog notation.

3.1.1 Boolean conjunctive queries (BCQs)

Additionally, we want to talk about a subclass of the CQs, namely the **boolean conjunctive queries (BCQs)**. BCQs have only "yes" or "no" as their output. There is no need for listing all tuples fulfilling the given conditions. It is sufficient to check if there is at least one tuple fulfilling these conditions. In many cases this simplifies the evaluation, since only one tuple needs to be found (if the answer is "yes").

For our example the question now can be if there is any professor, who is also a student. In datalog notation a BCQ has an empty head and for our example it looks like the following:

$$Q() : -S(N, C, S), P(P, C)$$

3.1.2 CQs in other domains

Since formally solving Constraint satisfaction problems (CSPs) is model-checking of first-order formulas, which contain only existence operators $\exists$ and logical ands $\wedge$, it can be considered equivalent to evaluating CQs. CSPs are widely used in different areas like in Artificial Intelligence (AI) and Operations Research. In AI and Machine Learning a general overview of CSPs was given by Tsang, 1993. Constraint Programming is broadly used as a method for solving CSPs. An overview is provided in Dechter, 2003. Among others Constraint Programming for Data Mining processes to solve CSPs is given by De Raedt et al., 2010 or for learning optimal decision trees by Verhaeghe et al., 2020. In Operations Research Constraint Programming was for instance used for Resource-Constrained Project Scheduling Problems by Geibinger et al., 2019 and for solving vehicle routing plans by Backer et al., 2000.

Therefore, we want to point out that optimizing the evaluation of CQs is not only useful in database theory applications, but also can influence topics in the domains AI and Operations research.

3.2 Complexity of evaluating CQs

Since CQs are very simple queries as explained above, e.g. in SQL they are SELECT-FROM-WHERE statements only allowing equality conditions connected with ANDs, one could assume that evaluating queries of this class should be easy. But Chandra and Merlin, 1977 showed, that they are even NP-complete. This means in the worst case the evaluation time of CQs increases exponentially with the size of the query.

In the last decades a lot of research was done to find classes of CQs, which still can be evaluated in polynomial time and there have been successes. Some of them will be introduced in the next chapters of this thesis (for example see Section 4.4 and 5.5).

CHAPTER 4

Hypergraphs and acyclicity

In Section 4.1 we introduce hypergraphs, which can be seen as abstractions of CQs. With this concept, we can use the queries' structure on a higher level. In Section 4.2 we explain join trees and how they correspond to hypergraphs. Together with the definition of join trees we can also introduce acyclic queries, a subclass of CQs. If a query has a join tree and therefore is acyclic, can be checked with the GYO-reduction algorithm, which we present in Section 4.3. Finally, after knowing a query is acyclic, it can be evaluated in polynomial time using Yannakakis' algorithm, which is given in Section 4.4.

4.1 Hypergraphs

This section is used to introduce hypergraphs, as well as explaining the connection between hypergraphs and CQs. Since a hypergraph is based on a graph, we first formally define what a graph is.

Definition 4.1.1 (Graph)**.** A *graph* $\mathcal{G}(V, E)$ consists of a set of vertices V and a set of edges E. The edges are connections between two vertices and can be written as tuples of vertices: $E \subseteq \{(v_1, v_2) | v_1, v_2 \in V\}$.

A hypergraph is a graph with edges, which are then called hyperedges, where each of them can also have more than two vertices. Formally a hypergraph can be defined as the following.

Definition 4.1.2 (Hypergraph)**.** A *hypergraph* $\mathcal{H}(V, E)$ consists of a set of vertices V and a set of hyperedges E. Each hyperedge is a non-empty set of vertices: $E \subseteq 2^V \setminus \{\emptyset\}$.

The structure of a CQ is crucial for properties like detecting acyclicity and the possibility to evaluate it efficiently (for an explanation see Section 4.2). Therefore, a hypergraph is

a useful abstraction of the CQ since it reflects its structure. The edges of the hypergraph correspond to the atoms of the CQ and the variables occurring in an atom of the CQ correspond to the vertices in an edge of the hypergraph. Therefore, the structure of the CQ is exactly represented by the hypergraph. Only the projections are neglected when using the hypergraph representation.

As an example we use the following query.

$$Q(F,R) :- \texttt{program(S,L,F)}, \texttt{student(N,I,S)}, \texttt{masterthesis(T,I,C)}, \texttt{professor(P,C)}, \texttt{room(R,C)}$$

The corresponding hypergraph is shown in Figure 4.1. The hypergraph $\mathcal{H}(V,E)$ has the vertices $V = \{L, F, S, N, I, T, C, P, R\}$ and the edges $E = \{\text{program}, \text{student}, \text{masterthesis}, \text{professor}, \text{room}\} = \{\{L,F,S\}, \{S,N,I\}, \{I,T,C\}, \{C,P\}, \{P,R\}\}$.

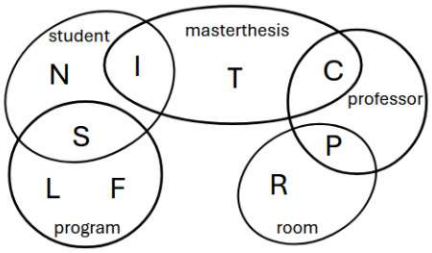


Figure 4.1: Example hypergraph.

As we said before, we can see that the structure of this query is represented by the hypergraph, but the final projection on F and R is not represented.

From now on we use hypergraphs and CQs as equivalent terms and we should always keep in mind that they represent the same underlying structure.

4.2 Acyclicity and join trees

In this section we want to introduce acyclicity and we are defining it using join trees. For summaries about acyclicity see Brault-Baron, 2016 and Fagin, 1983. Therefore, we introduce join trees first. The formal definition of a tree can be done like the following.

Definition 4.2.1 (Tree)**.** A *rooted tree* $\mathcal{T}(N,L)$ is a connected, acyclic graph with a root node. The set of nodes (= vertices) of a tree is denoted as N and the set of links (= edges) as L.

We use the terms nodes/links for trees and vertices/edges for graphs for easy distinction. Each node in the tree can have multiple children and exactly one parent node corresponding to the hierarchy. The root node is an exception since it has no parent. The leaf nodes are the ones with no children.

Using the definition of a tree and a hypergraph we can define join trees now.

Definition 4.2.2 (Join tree)**.** A *join tree* $\mathcal{T}(N, L)$ of a hypergraph $H(V, E)$ is a tree which fulfills the following conditions:

- $N = H(E)$
- $\forall v \in H(V)$: $\{e \in H(E) | v \in e\}$ induces a connected subtree of $\mathcal{T}$.

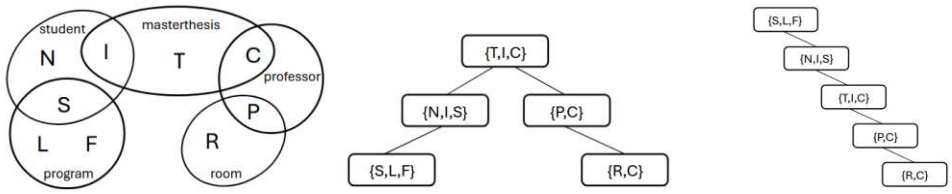
The second condition of a join tree's definition is often called "connectedness condition". For one hypergraph there can be multiple join trees with different roots. This is used in Section 5.5 for aggregate queries.

The definition of a join tree goes hand in hand with the definition of α-acyclicity of a hypergraph.

Definition 4.2.3 (Acyclicity)**.** A hypergraph $H(V, E)$ is *α-acyclic* if it has a join tree.

In the following, when we use the term acyclic, we mean α-acyclicity (see Fagin, 1983, Brault-Baron, 2016). An acyclic conjunctive query (ACQ) is a CQ, whose corresponding hypergraph is acyclic.

In Figure 4.2 we can see two join trees corresponding to the example hypergraph in Section 4.1, which we plot here again for better comparability.

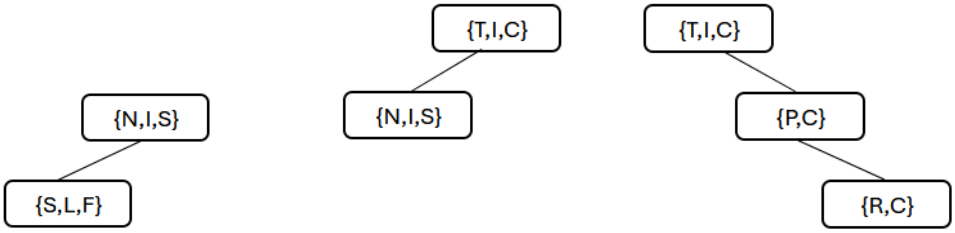


(a) Hypergraph. (b) Join tree. (c) Join tree.

Figure 4.2: Example of corresponding hypergraph and join trees.

This hypergraph is acyclic since it has a corresponding join tree. The join trees $\mathcal{T}(N, L)$ have the nodes $N = H(E) = \{\{S, L, F\}, \{N, I, S\}, \{T, I, C\}, \{P, C\}, \{R, C\}\}$ and unnamed links in between them. The difference between the join trees in our example is, that they have different root nodes. For join tree 4.2b the node $\{T, I, C\}$ is the root, whereas join tree 4.2c has $\{S, L, F\}$ as its root. Additionally, the connectedness condition can be checked. It can be seen that the variables L, F, N, T, P and R only occur once. Therefore, they fulfill the connectedness condition trivially. For the variables S, I and C we have to check if all hyperedges they appear in are connected. This is the case and we illustrate the connected subtrees for join tree 4.2b in Figure 4.3.

Note that a join tree can be seen as query plan. The joins can be executed from the bottom of the tree up to the root node.



(a) Connected subtree for S. (b) Connected subtree for I. (c) Connected subtree for C.

Figure 4.3: Illustration of the connectedness condition.

4.2.1 Complexity of evaluating ACQs

Yannakakis, 1981 introduced an algorithm enabling efficient evaluation of ACQs, demonstrating that their evaluation is tractable. This is a remarkable result, since we heard in Section 3.2 that the evaluation of CQs is NP-complete. The complexity of evaluating ACQs is bounded by $\widetilde{O}\left(||Q|| * ||D|| + ||Q(D)||\right)$, where $\widetilde{O}$ is the complexity hiding a log factor, $||D||$ represents the size of the input database instance, $||Q||$ the size of the ACQ Q and $||Q(D)||$ the output size of the query (Grohe et al., 2001). This complexity is often called total polynomial time or output polynomial time. It's important to note that although the algorithm's time complexity is polynomial, the output size can still be exponential in relation to the input size.
For boolean ACQs Gottlob et al., 1998 showed that the evaluation is even LOGFCL-complete. This means boolean ACQs can not only be evaluated in polynomial time, but are highly parallelizable.
Even if this is a favorable property, only a few real world queries are acyclic. Nevertheless, we can use this efficient evaluation of ACQs as basic case, which can be extended to bigger classes of queries via various notions of decompositions (see Section 4.4 and Section 5.5).

4.3 GYO-reduction

To check if a query has a join tree and if so to build one, we can use the GYO-reduction algorithm, which was introduced by Graham, 1979 and C. Yu and Ozsoyoglu, 1979 at the same time. With this algorithm a join tree can be computed efficiently if one exists. The procedure works as defined in Algorithm 4.1.

In words, the GYO-reduction applies two rules on a hypergraph exhaustively until no longer possible. The first rule is that hyperedges, which do not share any vertex with another hyperedge, are eliminated. The second rule is that hyperedges are eliminated if there is a witness for them. This means each vertex of the hyperedge is either exclusively part of the hyperedge itself or present in another hyperedge, which is then called "witness". Using these witnesses, we also get a join tree corresponding to the hypergraph. In this join tree every witness of a hyperedge is a parent node of the node of the hyperedge. The

Algorithm 4.1: GYO-reduction

Input: A hypergraph $\mathcal{H}(V, E)$
Procedure: GYO-REDUCTION($\mathcal{H}(V, E)$)
while *possible* **do**
  if $\exists e \in E : \forall v \in e : \nexists e' \in E \setminus \{e\} : v \in e'$ **then**
    delete e
  end
  if $\exists e \in E : \exists w \in E \setminus \{e\} : \forall v \in e : (v \in w \vee \nexists e' \in E \setminus \{e\} : v \in e')$ **then**
    assign w as witness of e and delete e
  end
end
return E;

hyperedges which only contain vertexes part of the hyperedge itself can be added to any node of the join tree. If the result of the GYO-reduction is empty, a join tree exists.

We illustrate one possible GYO-reduction using the hypergraph which we also used in the last two sections. It has the following vertices $V = \{L, F, S, N, I, T, C, P, R\}$ and hyperedges $E = \{\{S, L, F\}, \{N, I, S\}, \{T, I, C\}, \{P, C\}, \{R, C\}\}$ and is represented again in Figure 4.4.

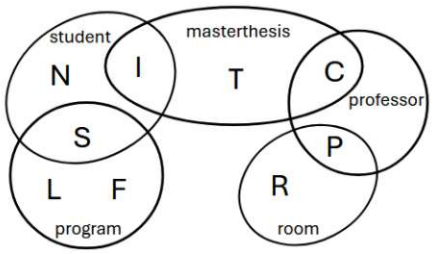


Figure 4.4: Example hypergraph.

We start with this set of hyperedges: $E = \{\{S, L, F\}, \{N, I, S\}, \{T, I, C\}, \{P, C\}, \{R, C\}\}$. As a first step we check if there is a hyperedge which does not share any vertex with another one. There is no such case. Therefore, we next eliminate a hyperedge which has a witness. We decide to take $\{S, L, F\}$, which has $\{N, I, S\}$ as witness. The new set looks like this: $E = \{\{N, I, S\}, \{T, I, C\}, \{P, C\}, \{R, C\}\}$. Again there is no hyperedge, which does not share any vertex with another hyperedge. So, we go to step two again and eliminate $\{N, I, S\}$ with witness $\{T, I, C\}$. This gives us: $E = \{\{T, I, C\}, \{P, C\}, \{R, C\}\}$. The first check again tells us, that there is no such hyperedge to eliminate. Next we eliminate $\{R, C\}$ and assign $\{P, C\}$ as witness. Be aware that in this case also $\{T, I, C\}$ could be the witness. We get this set: $E = \{\{T, I, C\}, \{P, C\}\}$. We again cannot eliminate a hyperedge with step one. The next elimination is $\{P, C\}$ with witness $\{T, I, C\}$. Now only this set remains: $E = \{\{T, I, C\}\}$. Therefore, the remaining hyperedge does not share any vertex with another hyperedge and we eliminate it, which gives this set: $E = \{\}$.

Now both rules cannot be applied anymore. Since the result is an empty set, we know that a join tree exists and that the hypergraph is acyclic. Moreover, we know the exact join tree using the witnesses. Since the GYO-reduction can give us different results due to the decisions in which order we eliminate the hyperedges and which witnesses we choose, there is no unique join tree, but several possible ones. The one corresponding to our example is the same one we already showed in Figure 4.2b.

4.4 Yannakakis' algorithm

Once we have a join tree for a query, we can apply Yannakakis' algorithm introduced by Yannakakis, 1981. It consists of three stages, the first bottom-up traversal, the top-down traversal and the second bottom-up traversal. The idea is to use only semi-joins to delete all dangling tuples, which are tuples that do not contribute to the final result and then only join the remaining, necessary tuples. By using this process for the evaluation of acyclic queries, we get a tractable evaluation time (for details see Chapter 4.2.1).

This algorithm can only be applied if the hypergraph is acyclic. Therefore, we also can construct a corresponding join tree using the GYO-algorithm. We take one join tree and the relations corresponding to each node of the join tree and these are the inputs to the algorithm.

The first bottom-up traversal performs semi-joins from the bottom up the tree. This means for each node in the join tree, which has at least one child node, we apply semi-joins on this node with each of its children starting at the bottom of the tree. After this traversal each node contains those tuples, that can be extended to form joins with all child nodes in the whole subtree.

The top-down traversal does exactly the same as the first bottom-up except that it starts at the top and performs semi-joins with the corresponding parent node for each node except the root. At this point, after the first two traversals, all dangling tuples have been deleted.

This leads us directly to the final stage. We can join all relations now and by construction all tuples, which are part of these joins, will be needed for the result, too. Therefore, we handled all unnecessary intermediate results in a way that there are none of them anymore and additionally we get a good join order. This traversal is again done bottom-up and the relation at the root node contains the result of the query in the end.

To ensure that the result of Yannakakis' algorithm (using the traversals and semi-joins) is the required result of the query, we provide the following correctness conditions (Eq. 4.1, 4.2, 4.3). For the used join tree $\mathcal{T}(N, L)$ we denote a subtree of $\mathcal{T}$ rooted at the node $n \in N$ as $\mathcal{T}_n(N_n, L_n)$. The relation corresponding to a node $n \in N$ is called R_n and depending on the result of the first, second and third traversal the corresponding relations are denoted as R'_n, R''_n and R'''_n respectively. The columns of a relation R_n at the node n are denoted as $col(n)$ and the columns of all joined relations of a subtree $\mathcal{T}_n(N_n, L_n)$ at the node n are denoted as $col(N_n)$.

After the first traversal each relation has the following form

$$\forall n \in N : R_n^{'} = \pi_{col(n)}(\bowtie_{m \in N_n}) R_m \tag{4.1}$$

where $(\bowtie_{m \in N_n}) R_m$ means, that the relations of all nodes of the subtree $\mathcal{T}_n(N_n, L_n)$ are joined together. This equation tells us that after the first traversal each relation contains the same values as if we would have performed joins in the subtree of that relation and projected on the columns of that relation.
After the second traversal we have the results of joins in the whole tree projected on the columns of the relation we look at.

$$\forall n \in N : R_n^{''} = \pi_{col(n)}(\bowtie_{m \in N}) R_m \tag{4.2}$$

In the end, after the third traversal each relation has those values, which correspond to joining all relations and projecting on the relations of the whole subtree.

$$\forall n \in N : R_n^{'''} = \pi_{col(N_n)}(\bowtie_{m \in N}) R_m \tag{4.3}$$

Using these correctness conditions, we directly get that after the last traversal the root contains the result of the query. Equation 4.3 looks like that $R_r^{'''} = \pi_{col(N_r)}(\bowtie_{m \in N}) R_m = \pi_{col(N)}(\bowtie_{m \in N}) R_m = (\bowtie_{m \in N}) R_m$ for the root node r. Therefore we get the overall join result of all tables (projected on all columns of all relations, which is equal to no projection).

In Algorithm 4.2 we provide the pseudo-code for Yannakakis' algorithm. In lines 2-17 we introduce boolean flags to ensure that the join order is indeed bottom-up or top-down, depending on the stage. For the two bottom-up traversals we only perform (semi-)joins on those nodes, which have child nodes, and for the top-down only for the ones, which have a parent. This is ensured in lines 19-21. The first bottom-up traversal is given in lines 22-28, followed by the top-down traversal in lines 29-33 and in lines 34-38 the second bottom-up traversal is performed. As result we return the final relation at the root node, which now contains the query result by construction.

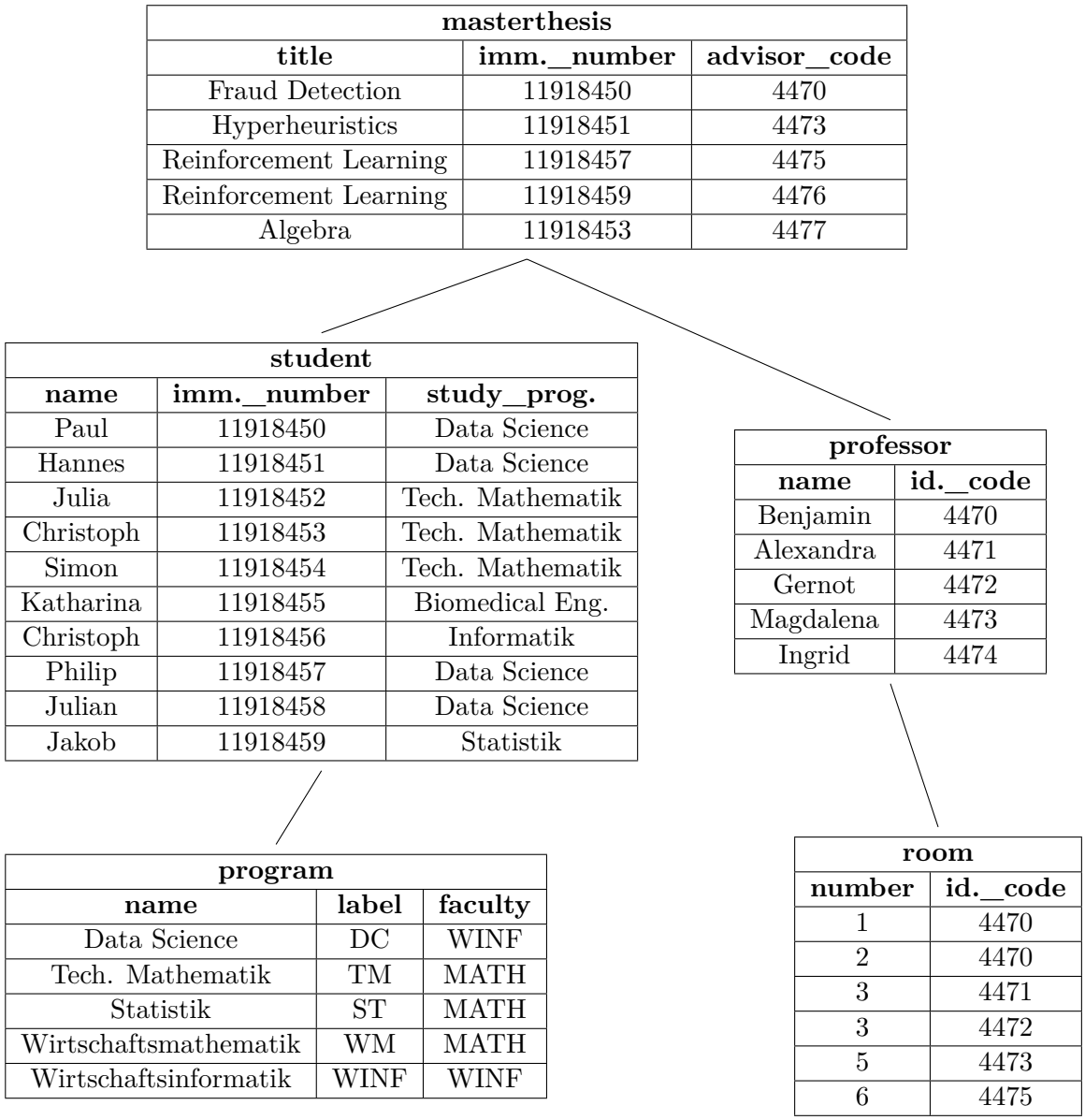
Furthermore, evaluating BCQs even gets easier (assuming that they are acyclic) and less complex (for details see Chapter 4.2.1). For the evaluation of BCQs the first bottom-up traversal is sufficient. In this traversal, we check if at least one tuple of the root node can be extended to form a join with all other nodes. Using Equation 4.1 for the root node r we get $R_r^{'} = \pi_{col(r)}(\bowtie_{m \in N_r}) R_m = \pi_{col(r)}(\bowtie_{m \in N}) R_m$ and we can see that we get the join result of all relations projected on the columns of the relation at the root node. For the BCQ it is sufficient to know if there is or if there is not at least one tuple in the result. This behaviour can be used for 0MA queries as well, which are explained in Chapter 5.5.

To show how Yannakakis' algorithm works, we give an example using the database schema we defined in 3.1. The query we want to evaluate is the same we used in the last sections. We showed in Section 4.3 that it is acyclic and Yannakakis' algorithm can be applied. We will use the join tree we provided before and refer to it as "the" join tree in the rest of this section, but be aware that we could also use different join trees.

Algorithm 4.2: Yannakakis' algorithm

Input: A join tree $\mathcal{T}(N, L)$ and $\forall n \in N$ a corresponding relation R_n, the set of child nodes $C(n) = \{c_1(n), ..., c_k(n)\}$ and the parent node $p(n)$

Procedure: YANNAKAKIS($\mathcal{T}(N, L)$)
for $n \in N$ **do**
  if *n is a leaf node* **then**
    $up1\_flag_n$ = TRUE
    $down\_flag_n$ = FALSE
    $up2\_flag_n$ = TRUE
  end
  else if *n is the root node* **then**
    $up1\_flag_n$ = FALSE
    $down\_flag_n$ = TRUE
    $up2\_flag_n$ = FALSE
  end
  else
    $up1\_flag_n$ = FALSE
    $down\_flag_n$ = FALSE
    $up2\_flag_n$ = FALSE
  end
end
$NC1 := \{n \in N | C(n) \neq \emptyset\}$
$NP := \{n \in N | \exists\, p(n)\}$
$NC2 := \{n \in N | C(n) \neq \emptyset\}$
while $\exists n \in NC1 : \forall c_i(n) \in C(n) : up1\_flag_n = TRUE$ **do**
  for $i \in \{1, ..., k\}$ **do**
    $R_n := R_n \ltimes R_{c_i(n)}$
  end
  $up1\_flag_n$ = TRUE
  $NC1 := NC1 \setminus \{n\}$
end
while $\exists n \in NP : down\_flag_{p(n)} = TRUE$ **do**
  $R_n := R_n \ltimes R_{p(n)}$
  $down\_flag_n$ = TRUE
  $NP := NP \setminus \{n\}$
end
while $\exists n \in NC2 : \forall c_i(n) \in C(n) : up2\_flag_n = TRUE$ **do**
  $R_n := R_n \bowtie R_{c_i(n)}$
  $up2\_flag_n$ = TRUE
  $NC2 := NC2 \setminus \{n\}$
end
return R_r = *the relation of the root node;*



masterthesis		
title	**imm._number**	**advisor_code**
Fraud Detection	11918450	4470
Hyperheuristics	11918451	4473
Reinforcement Learning	11918457	4475
Reinforcement Learning	11918459	4476
Algebra	11918453	4477

student		
name	**imm._number**	**study_prog.**
Paul	11918450	Data Science
Hannes	11918451	Data Science
Julia	11918452	Tech. Mathematik
Christoph	11918453	Tech. Mathematik
Simon	11918454	Tech. Mathematik
Katharina	11918455	Biomedical Eng.
Christoph	11918456	Informatik
Philip	11918457	Data Science
Julian	11918458	Data Science
Jakob	11918459	Statistik

professor	
name	**id._code**
Benjamin	4470
Alexandra	4471
Gernot	4472
Magdalena	4473
Ingrid	4474

program		
name	**label**	**faculty**
Data Science	DC	WINF
Tech. Mathematik	TM	MATH
Statistik	ST	MATH
Wirtschaftsmathematik	WM	MATH
Wirtschaftsinformatik	WINF	WINF

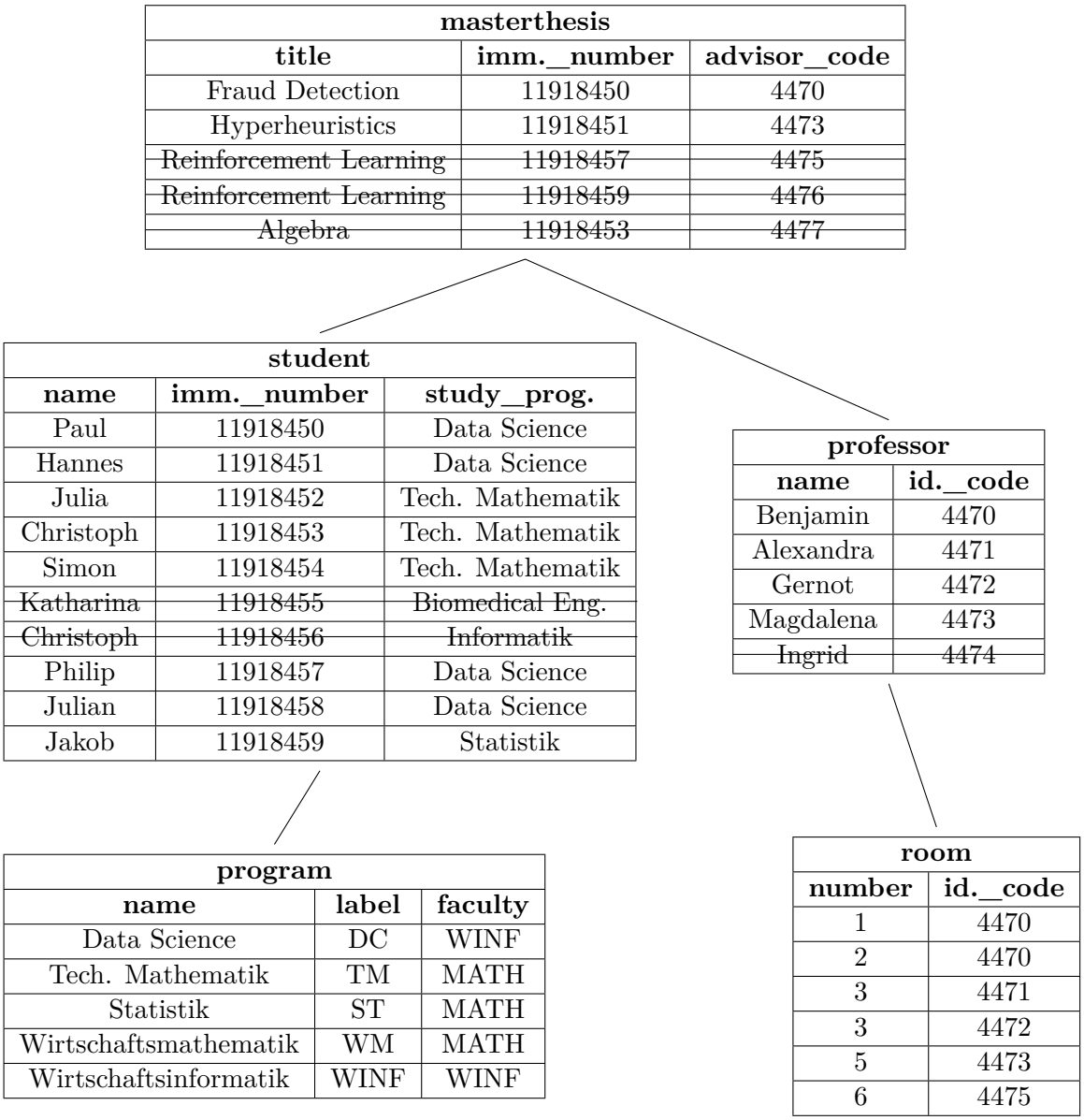
room	
number	**id._code**
1	4470
2	4470
3	4471
3	4472
5	4473
6	4475

Figure 4.5: Example join tree with instances.

The query looks like the following:

$$Q(F,R) :- \texttt{program(S,L,F)}, \texttt{student(N,I,S)}, \texttt{masterthesis(T,I,C)}, \texttt{professor(P,C)}, \texttt{room(R,C)}$$

Before starting with the algorithm, we have to define instances for our relations. We provide them together with the schematic representation of the join tree in Figure 4.5.



masterthesis		
title	**imm._number**	**advisor_code**
Fraud Detection	11918450	4470
Hyperheuristics	11918451	4473
~~Reinforcement Learning~~	~~11918457~~	~~4475~~
~~Reinforcement Learning~~	~~11918459~~	~~4476~~
~~Algebra~~	~~11918453~~	~~4477~~

student		
name	**imm._number**	**study_prog.**
Paul	11918450	Data Science
Hannes	11918451	Data Science
Julia	11918452	Tech. Mathematik
Christoph	11918453	Tech. Mathematik
Simon	11918454	Tech. Mathematik
~~Katharina~~	~~11918455~~	~~Biomedical Eng.~~
~~Christoph~~	~~11918456~~	~~Informatik~~
Philip	11918457	Data Science
Julian	11918458	Data Science
Jakob	11918459	Statistik

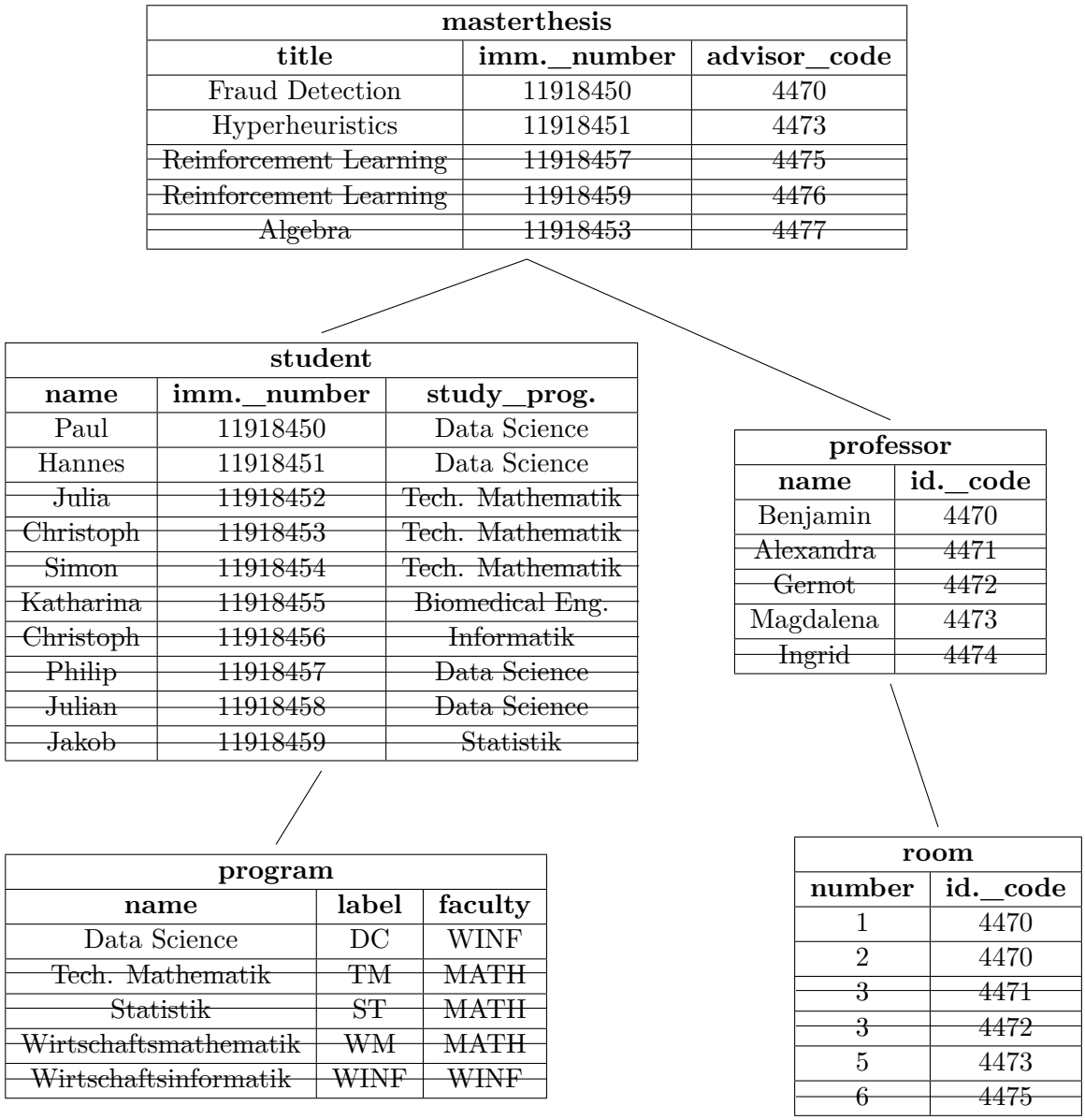
professor	
name	**id._code**
Benjamin	4470
Alexandra	4471
Gernot	4472
Magdalena	4473
~~Ingrid~~	~~4474~~

program		
name	**label**	**faculty**
Data Science	DC	WINF
Tech. Mathematik	TM	MATH
Statistik	ST	MATH
Wirtschaftsmathematik	WM	MATH
Wirtschaftsinformatik	WINF	WINF

room	
number	**id._code**
1	4470
2	4470
3	4471
3	4472
5	4473
6	4475

Figure 4.6: Example join tree with instances after the first top-down traversal.

After the first bottom-up traversal, which performs semi-joins beginning at the bottom and then going up the tree, some tuples have been deleted. The relations look like in Figure 4.6 now. To get this result we take the relation `student` and perform a semi-join with `program`, which only leaves those tuples that find a join partner. The same is done for relation `professor` with `room`. Afterwards semi-joins between `masterthesis` and `student` and between `masterthesis` and `professor` are performed.



masterthesis		
title	**imm._number**	**advisor_code**
Fraud Detection	11918450	4470
Hyperheuristics	11918451	4473
~~Reinforcement Learning~~	~~11918457~~	~~4475~~
~~Reinforcement Learning~~	~~11918459~~	~~4476~~
~~Algebra~~	~~11918453~~	~~4477~~

student		
name	**imm._number**	**study_prog.**
Paul	11918450	Data Science
Hannes	11918451	Data Science
~~Julia~~	~~11918452~~	~~Tech. Mathematik~~
~~Christoph~~	~~11918453~~	~~Tech. Mathematik~~
~~Simon~~	~~11918454~~	~~Tech. Mathematik~~
~~Katharina~~	~~11918455~~	~~Biomedical Eng.~~
~~Christoph~~	~~11918456~~	~~Informatik~~
~~Philip~~	~~11918457~~	~~Data Science~~
~~Julian~~	~~11918458~~	~~Data Science~~
~~Jakob~~	~~11918459~~	~~Statistik~~

professor	
name	**id._code**
Benjamin	4470
~~Alexandra~~	~~4471~~
~~Gernot~~	~~4472~~
Magdalena	4473
~~Ingrid~~	~~4474~~

program		
name	**label**	**faculty**
Data Science	DC	WINF
~~Tech. Mathematik~~	~~TM~~	~~MATH~~
~~Statistik~~	~~ST~~	~~MATH~~
~~Wirtschaftsmathematik~~	~~WM~~	~~MATH~~
~~Wirtschaftsinformatik~~	~~WINF~~	~~WINF~~

room	
number	**id._code**
1	4470
2	4470
~~3~~	~~4471~~
~~3~~	~~4472~~
5	4473
~~6~~	~~4475~~

Figure 4.7: Example join tree with instances after the bottom-up traversal.

Similarly semi-joins for the top-down traversal are performed. Note that only tuples that have been left after the first traversal are used now. The semi-joins here are done between `student` and `masterthesis`, between `professor` and `masterthesis`, between `program` and `student` and between `room` and `professor`. In Figure 4.7 we can see the resulting relations without dangling tuples as a result after the bottom-up traversal.

Finally, in the second bottom-up traversal joins between the relations having only tuples left, which are contributing to the result, are performed, again starting at the bottom. In this case it would also work to do a top-down traversal. Nevertheless be aware that a random order would not work, since most of the tables do not share a join column then. For our example in this final traversal we start with joining `student` on attribute "study_prog." with the attribute "name" of the `program` relation. This join result is then joined on "imm._number" with `masterthesis` on the "imm._number" attribute. We call that intermediate result `masterthesis1` for now. After that we join `professor` with `room` on "id._code", which is then joined with `masterthesis1` on "advisor_code". The result of Yannakakis' algorithm is therefore one table consisting of all columns and the remaining rows at the root node. For our example, the result is presented in Table 4.1.

Join result								
T	**N**	**I**	**S**	**L**	**F**	**P**	**C**	**R**
Fraud Detection	Paul	11918450	Data Science	DC	WINF	Benjamin	4470	1
Fraud Detection	Paul	11918450	Data Science	DC	WINF	Benjamin	4470	2
Hyperheuristics	Hannes	11918451	Data Science	DC	WINF	Magdalena	4473	5

Table 4.1: Join result of Yannakakis' algorithm.

In the Figures 4.5, 4.6, 4.7 we used recognizable abbreviations for some of the attribute names. In this result table we decided to only use single letters because of space issues again. These letters correspond to the letters used in the query we want to evaluate. "T" corresponds to "title" of the relation `masterthesis`. The "name" of the `student` is represented with "N". "I" is the "imm._number" (="immatriculation_number") of both `masterthesis` and `student`. The "study_prog." (= "study_program") of `student` was joined with "name" of `program` and called "S" now. "L" and "F" are the "label" and "faculty" of the `program`. The attribute "name" of `professor` is represented with "P". Then "C" is the "advisor_code" of `masterthesis` and the "id._code" (= "identification_code") of both `professor` and `room`. And "R" represents the attribute "number" of the relation `room`.

Yannakakis' algorithm is finished here, but the query still has a projection on the attributes "F" and "R". The final evaluation result of that query is given in Table 4.2.

Query result	
F	**R**
WINF	1
WINF	2
WINF	5

Table 4.2: Evaluation result of the example query.

CHAPTER 5

Decompositions and beyond CQs

In this chapter we introduce several decompositions of trees and graphs with their corresponding width measure. Decompositions can be useful to split complex structures and computational problems into smaller problems, which can be calculated more efficiently. Additionally, they can be used to transform cyclic queries into acyclic ones. The width notions are often used to quantify the complexity of graphs and problems.

In Section 5.1 we introduce tree decompositions, then in Section 5.2 hypertree decompositions (HDs) and in Section 5.3 generalized hypertree decompositions (GHDs). In Section 5.4 we consider the computational properties of finding and using decompositions, which includes connecting and comparing join trees, HDs and GHDs.

Decompositions and width measures are highly useful in our context, since Bonifati et al., 2017 and Fischl et al., 2021 analyzed millions of benchmark queries and query logs, where nearly every query of those is "almost acyclic" in a sense of having HDs width smaller or equal to two. Similar methods as for acyclic queries can be applied to "almost acyclic" queries, which leads to efficient evaluation (Gottlob et al., 1999).

The other part of this chapter is about going beyond CQs. Even if CQs are fundamental queries and are present in real-world applications, this class of queries is very restricted. Therefore, we want to allow some additional properties to broaden the class of queries we consider. In Section 5.5 we look at CQs with additional aggregates and introduce the zero-materialisation answerable (0MA) queries from Gottlob et al., 2023.

5.1 Tree decompositions

The first decomposition of a graph, which we consider, is the tree decomposition with the tree width introduced by Robertson and Seymour, 1983. Formally we can define tree decompositions like in the following using trees and graphs like in Definition 4.2.1 and Definition 4.1.1.

Definition 5.1.1 (Tree Decomposition)**.** A *tree decomposition* $< \mathcal{T}, \chi >$ of a graph $\mathcal{G}(V, E)$ is a tree $\mathcal{T}(N, L)$ with a labelling function $\chi : n \in N \rightarrow \chi(n) \subseteq V$, which fulfills the following conditions:

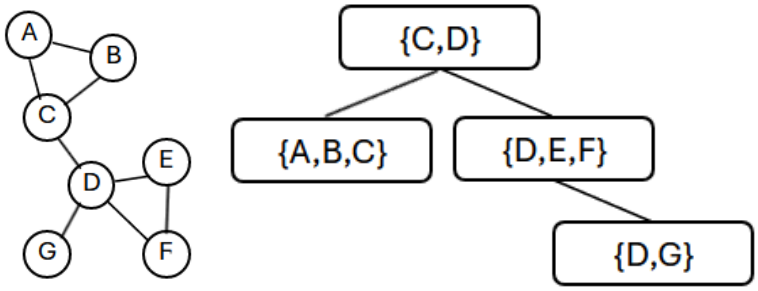
1. For each graph vertex $v \in V$ there exists a tree node $n \in N$: $v \in \chi(n)$.
2. For each graph edge $(v_1, v_2) \in E$ there exists a tree node $n \in N$: $(v_1, v_2) \subseteq \chi(n)$.
3. For each graph vertex $v \in V$ the set $\{n \in N | v \in \chi(n)\}$ induces a connected subtree $\mathcal{T}'$ of $\mathcal{T}$ (connectedness condition).

A tree decomposition is a tree, which corresponds to a graph and is connected through the labelling function. Each graph vertex has to be contained in a tree node and each graph edge has to be a subset of a tree node. The connectedness condition says that for each vertex the nodes in the tree where it appears have to be connected.

Now we can introduce the corresponding width measure of a graph, the tree width.

Definition 5.1.2 (Tree width)**.** The *width* of a tree decomposition $< \mathcal{T}, \chi >$ with $\mathcal{T}(N, L)$ is $\max_{n \in N}(|\chi(n)| - 1)$. The *tree width tw($\mathcal{G}$)* of a graph $\mathcal{G}$ is the minimum width over all its tree decompositions.

We give an example for a tree decomposition with width 2 in Figure 5.1.



(a) Graph. (b) Tree decomposition.

Figure 5.1: Example tree decomposition.

Each vertex A-G is contained in at least one node in the tree. Each edge $\{\{A, B\}, \{A, C\}, \{B, C\}, \{C, D\}, \{D, E\}, \{D, F\}, \{E, F\}, \{D, G\}\}$ is contained in one node in the tree and the connectedness condition is also fulfilled since the subgraphs for each vertex are connected.

5.2 Hypertree decompositions (HDs)

The hypertree decomposition (HD) is a generalization of the tree decomposition. Descriptions of the HD and width are given in Gottlob et al., 1999. We first have to introduce hypertrees.

Definition 5.2.1 (Hypertree)**.** A *hypertree* $<\mathcal{T}, \chi, \lambda>$ of a hypergraph $\mathcal{H}(V, E)$ is a rooted tree $\mathcal{T}(N, L)$ with two labeling functions χ and λ. The functions assign a set of vertices or edges for each tree node: $\chi : n \in N \rightarrow \chi(n) \subseteq V$ and $\lambda : n \in N \rightarrow \lambda(n) \subseteq E$.

With that definition we can formally define HDs now.

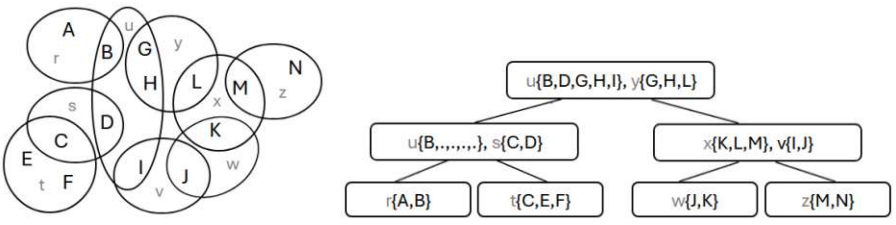
Definition 5.2.2 (Hypertree Decomposition (HD))**.** A *hypertree decomposition* $HD = < \mathcal{T}, \chi, \lambda >$ of a hypergraph $\mathcal{H}(V, E)$ is a hypertree with the rooted tree $\mathcal{T}(N, L)$ and the two labelling functions χ and λ, which fulfills the following conditions:

1. For each hypergraph edge $e \in E$ there exists a hypertree node $n \in N$: $e \subseteq \chi(n)$.
2. For each hypertree node $n \in N$: $\chi(n) \subseteq var(\lambda(n))$.
3. For each variable in the hypergraph $y \in var(\mathcal{H})$: $\{n \in N | y \in \chi(n)\}$ induces a connected subtree of $\mathcal{T}$ (connectedness condition).
4. For each hypertree node $n \in N$: $var(\lambda(n)) \cap \chi(\mathcal{T}_n) \subseteq \chi(n)$ (special condition).

In this definition $\mathcal{T}_n$ denotes the subtree of $\mathcal{T}$ rooted at n. So, the definition tells us, which conditions the hypertree of the hypergraph has to fulfill to be a HD. First, for each hypergraph edge there has to be a hypertree node, such that all variables of the hypergraph edge are covered by that node of the hypertree. For each hypertree node the vertices at that node are a subset of the variables assigned by the labelling function $\lambda(n)$, which gives the edges that cover this node. The third condition is the connectedness condition again, like we had for the tree decompositions. For the special condition for each hypertree node the variables assigned by $\lambda(n)$ intersected with the vertices of the subtree rooted at the node have to form a subset of the vertices at the node n. In other words if a vertex, which is covered by the edges that cover one node ($\lambda(n)$), does not appear in the bag of vertices of that node, it must not reappear anywhere in the subtree.

The corresponding hypertree width is defined as:

Definition 5.2.3 (Hypertree width)**.** The *width* of a hypertree $<\mathcal{T}, \chi, \lambda>$ with $\mathcal{T}(N, L)$ is $\max_{n \in N}(|\lambda(n)|)$.
The *hypertree width hw($\mathcal{H}$)* of a hypergraph $\mathcal{H}$ is the minimum width over all its hypertree decompositions.



(a) Hypergraph. (b) Hypertree decomposition.

Figure 5.2: Example hypertree decomposition.

To illustrate the HD we provide an example in Figure 5.2. We can see a hypergraph and its corresponding HD including the edge cover, which represents the labelling function λ. It gives the edges, which cover the vertices in the node.

The first condition is fulfilled, since all hypergraph edges: $\{\{A, B\}, \{C, D\}, \{C, E, F\}, \{B, D, G, H, I\}, \{I, J\}, \{J, K\}, \{K, L, M\}, \{M, N\}\}$ appear in one of the nodes of the hypergraph. For checking the second condition we can use the edge cover. For the node $\{B, C, D\}$ the $var(\lambda(n)) = \{B, D, G, H, I, C, D\}$, so the subset condition is fulfilled. For all other nodes it is fulfilled with equality. The connectedness condition can be checked vertex by vertex and it can be seen that they induce subtrees each time. For the special condition we have to take all variables in the edge cover and intersect it with all variables in the subtree for each node and we can see that they are indeed subsets of the variables at node n. For example for the node $\{B, C, D\}$ we get $\{B, D, G, H, I, C, D\} \cap \{A, B, C, E, F\} = \{B, C\} \subset \{B, C, D\}$.

But if for example the node $\{A, B\}$ would be $\{A, B, G\}$ this condition would not be fulfilled ($\{B, D, G, H, I, C, D\} \cap \{A, B, G, C, E, F\} = \{B, G, C\} \not\subset \{B, C, D\}$).
The width of this HD is 2.

5.3 Generalized hypertree decompositions (GHDs)

The generalized hypertree decomposition (GHD) is a further generalization of the hypertree decomposition. Insights can be gained in Gottlob et al., 2011, as well as comparisons of the different width measures in Gottlob et al., 2016.
We introduce GHDs formally.

Definition 5.3.1 (Generalized Hypertree Decomposition (GHD))**.** A *hypertree decomposition* $GHD =< \mathcal{T}, \chi, \lambda >$ of a hypergraph $\mathcal{H}(V, E)$ is a hypertree with the rooted tree $\mathcal{T} = (N, L)$ and the two labelling functions χ and λ, which fulfills the following conditions:

1. For each hypergraph edge $e \in E$ there exists a hypertree node $n \in N$: $e \subseteq \chi(n)$.
2. For each hypertree node $n \in N$: $\chi(n) \subseteq var(\lambda(n))$.
3. For each variable in the hypergraph $y \in var(\mathcal{H})$: $\{n \in N | y \in \chi(n)\}$ induces a connected subtree of $\mathcal{T}$ (connectedness condition).

This definition is exactly the same as the hypertree decomposition definition, except that the last condition of the HD does not have to be fulfilled by a GHD.
The corresponding width is defined equivalently as before as:

Definition 5.3.2 (Generalized hypertree width)**.** The *width* of a hypertree $<\mathcal{T}, \chi, \lambda>$ with $\mathcal{T}(N, L)$ is $\max_{n \in N}(|\lambda(n)|)$.
The *generalized hypertree width ghw($\mathcal{H}$)* of a hypergraph $\mathcal{H}$ is the minimum width over all its GHD.

We do not provide another example here, since we have got the example for an HD above, which is also a GHD. Additionally, we mentioned before, that if the node $\{A, B\}$ would be $\{A, B, G\}$ the special condition would not be fulfilled, which means it would not be a HD anymore, but it would be a GHD.

5.4 Computational properties

In this section we want to provide computational properties of decompositions, which includes finding and using them. Additionally, this gives us connections and comparisons between the different decomposition concepts.

Starting with comparing HDs and GHDs, one could ask, why we should ever use HDs, since they restrict the class of GHDs. This is due to the complexity of computing the decompostitions. Given a CQ and computing if the width is smaller or equal a fixed number (mostly 2 or 3) is NP-complete for the generalized hypertree decomposition/width, but tractable for the hypertree decomposition/width. Additionally, Fischl et al., 2021 showed, that the hypertree width and the generalized hypertree width are the same for most real-world queries. This means hypertree decompostitions can be computed faster and are most of the time not worse than GHDs.

After getting a HD we would like to have the same favorable behaviour than join trees. This can be achieve very easily. We just have to join the relations in each node of the hypertree together and this gives us a join tree, which can then be processes as in Section 4.4 with Yannakakis' algorithm.

5.5 Beyond CQs: 0MA queries

The class of CQs is a basic class of queries, only allowing selections, projections and equi-joins. We now want to introduce a class of queries, where aggregates are allowed. As mentioned in Section 4.4 BCQs can be evaluated by using only one traversal of Yannakakis' algorithm. This decreases the evaluation time enormously. That is the reason why the **zero-materialisation answerable (0MA)** class was introduced recently in Gottlob et al., 2023. 0MA queries are queries, which can be answered after the first traversal of Yannakakis' algorithm. Naturally, BCQs are a subclass of 0MA, but additionally all queries with the aggregates MIN and MAX and aggregates in combination with DISTINCT, like COUNT(DISTINCT ...), are 0MA. To formally define the class of 0MA queries, we need some other definitions, starting with queries in aggregation normal form.

Definition 5.5.1 (Aggregation normal form of a query)**.** A query Q is in aggregation normal form $\Leftrightarrow$ it is of the form $\gamma_U(\pi_S(Q'))$, where Q' is a CQ.

In this definition γ_U represents a group-by of the attributes and aggregate expressions contained in U and π_S a projection on the attributes S. Since the group-by implicitly also projects on some attributes, the π_S would not be required formally.

Using this we can introduce guarded queries.

Definition 5.5.2 (Guarded query)**.** A query Q in aggregation normal form, i.e. $Q = \gamma_U(\pi_S(Q'))$, is guarded $\Leftrightarrow \exists$ relation R in Q': $Att(U) \subset Att(R)$.

Guarded therefore means, that all attributes in the group-by clause occur in the relation R. So R "guards" the query Q or is the "guard" of Q.

Now, we define what set-safe queries are.

Definition 5.5.3 (Set-safe query)**.** A query Q in aggregation normal form, i.e. $Q = \gamma_U(\pi_S(Q'))$, is set-safe $\Leftrightarrow Q = \gamma_U(\delta(\pi_S(Q')))$.

Here δ represents duplicate elimination. This means a query is set-safe if duplicate elimination before the group-by does not change the output of the query.

Now we have all we need to introduce 0MA queries formally.

Definition 5.5.4 (Zero-materialisation answerable/0MA query)**.** A query Q in aggregation normal form is zero-materialisation answerable (0MA) $\Leftrightarrow Q$ is guarded and set-safe.

Therefore, 0MA queries have one node containing all aggregation attributes. As we heard before hypergraphs have several join trees and we then choose the join tree with the node containing the attributes as root. This enables that only one traversal of Yannakakis' algorithm is sufficient.

A further extension would be to consider counting-based aggregate queries, which are guarded, but do not need the set-safety anymore. We are not considering them in this thesis, but this class is presented in Lanzinger et al., 2024.

CHAPTER 6

Machine Learning

In this chapter we want to explain some Machine Learning (ML) concepts such as the models, which we can use for our decision problem of the thesis. Machine Learning can be considered as part of Artificial Intelligence (AI), which has experienced a big hype in the last few years and still is. Machine Learning has three common sub-disciplines: Unsupervised (Machine) Learning, Supervised (Machine) Learning and Reinforcement Learning. An overview of Machine Learning in general, definitions and models are given in Mitchell, 1997 and Mohri et al., 2012.

Unsupervised Learning methods are applied on unlabelled data and the goal is to find clusters, groups, patterns or outliers in the data. A review of different unsupervised methods is given by Naeem et al., 2023. One of the most common tools of unsupervised ML is clustering, about which Rodriguez et al., 2016 give an overview.
Supervised Learning works with labelled data, which means a response variable is given for the training data. The goal is to design a model, which is able to predict the label for unseen data. Overviews of some supervised ML techniques are given by Nasteski, 2017 and Alloghani et al., 2020. A comparison of different supervised ML methods on real-world data is done by Crisci et al., 2012.
In *Reinforcement Learning* methods an agent takes actions in a given environment and learns a policy in order to maximize the cumulative reward. An introduction to Reinforcement Learning is provided by Sutton and Barto, 2018.

We are going to use Supervised Machine Learning, since we are dealing with labelled data. Supervised ML is divided into two big concepts: Classification and Regression. **Classification** methods are used, when the response variable has values of a predefined set of classes, whereas the response variable for **Regression** tasks is continuous. In Choudhary and Gianey, 2017 an overview of regression methods and classification methods is provided. Caruana and Niculescu-Mizil, 2006 give an empirical comparison of eight supervised ML models on eleven different binary classification datasets. Almaghrebi et al., 2020 compare three regression models on a prediction task.

We start with describing several supervised learning methods in Section 6.1 for classification and regression. Additional to the models that are used other things like data preparation and evaluation are crucial tasks for ML. We will describe the concept of data augmentation in Section 6.2, which helps creating data if there are not enough training samples. In Section 6.3 we then provide the experiment design of a model selection workflow to be able to compare the different models.

6.1 Supervised Learning Models

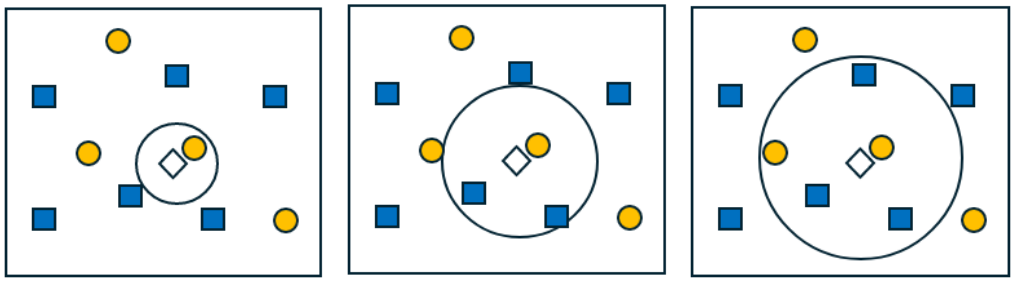
There is a wide range of available ML models and we decided to use and describe six of them (plus one combination of two of these). In Section 6.1.1 we introduce the k-Nearest Neighbors (k-NN) model, which is a lazy learner, but easy interpretable. In Section 6.1.2 we explain decision trees, which are easy to understand and interpret, but sometimes are prone to overfitting. Therefore, in Section 6.1.3 we introduce random forests, which are a combination of several decision trees to avoid the overfitting, but they are computationally more expensive. For high dimensional data support vector machines are very useful, but they only can be applied for binary problems. We will explain them in Section 6.1.4. In Section 6.1.5 multi-layer perceptrons are introduced, which can be considered as deep learning models. They are highly flexible and powerful, but need a lot of training data and are often not interpretable. Similarly, we use a hypergraph neural network, which is a deep learning model, but takes a hypergraph as input instead of numerical features (Section 6.1.6). To combine the information of the numerical features and the hypergraph, we design a neural network, which combines two multi-layer perceptrons and the hypergraph neural network in Section 6.1.7. These models can all be applied for classification and regression with some modifications.

6.1.1 k-Nearest Neighbors (k-NN)

One of the most straightforward ML models is the k-Nearest Neighbour algorithm. The k-NN is mostly used for classification, which is why we start to explain the k-NN classifier. Afterwards we will extend it to regression.

Given a new instance the k-NN classifier searches for the k nearest training set instances based on a chosen distance metric and takes the majority class of those neighbors as prediction. This very easy approach is a big advantage and k-NNs are often used as first model applied to a task. Nevertheless, the k-NN model is a lazy learner. This means it calculates everything at that time, when the new instance is given to the algorithm. So, if we have a lot of data we want to get predictions for, the computation is done at the classification step for each single instance, since there is no training phase before. This can lead to very long running times. Also, depending on the size of the training set, the running times can increase enormously, since every new instance is compared to every instance in the training set to get the nearest ones. Moreover, the choice of the parameter k and the used distance measure are crucial for the result. The k-NN model is

highly sensitive to local noise and not robust. An overview of k-NN classifiers is given in Cunningham and Delany, 2007.

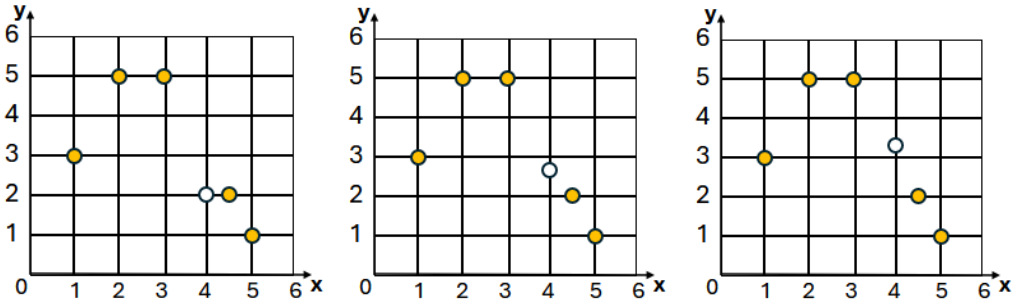


(a) k = 1. (b) k = 3. (c) k = 5.

Figure 6.1: Illustration of the k-NN classifier with two classes, Euclidean distance and different values of k.

To illustrate the k-NN classifier we provide Figure 6.1. For this example we decided to use the euclidean distance and two classes, which are yellow circles or blue rectangles. For a given new instance, the white diamond, we want to predict the class for k = 1, k = 3 and k = 5. It is common to use k equal to a number, which cannot be divided by the number of classes to avoid ties. In Figure 6.1a we can see that the predicted class would be circle/yellow, but for the other two examples in Figure 6.1b and 6.1c rectangle/blue would be the prediction.

For a k-NN regression the k nearest neighbors based on the given features are taken and the average (or weighted average) of their values is the new predicted value. In Figure 6.2 we show a simple regression example, where x is the given feature and y should be predicted. We want to predict the value at x = 4. For k = 1 we get the value 2, for k = 3 we get 2.67 and for k = 5 the value is 3.2.



(a) k = 1. (b) k = 3. (c) k = 5.

Figure 6.2: Illustration of a simple k-NN regressor using the average and different values of k.

6.1.2 Decision Tree

Decision trees are models with resulting trees (see Definition 4.2.1), where each inner node represents a decision and the leaf nodes the prediction. Decision trees are highly explainable and therefore widely used. They can be applied to both classification and regression and additionally for a mixture of them. At each node there is an equality or inequality condition for a feature with a value/class. It is also possible that multiple conditions for the same or different features are given at one node. This divides the feature space in sections, where each section gets an assigned label. After the training process the decision tree gets a new instance and just has to check the conditions at the nodes and assign the label or value at the leaf node, where the instance belongs to. An introduction of decision trees is given by Quinlan, 1986.

A decision tree tries to separate all instances in the training set by setting conditions based on the values of the features, such that each training instance is correctly classified. If some instances are similar, but the predictions different, the decision tree tries to distinguish the features further and further. Therefore, decision trees are prone to overfitting.

Instances	Age	Sex	Education	Married
Training sample 1	20	F	Primary	Y
Training sample 2	20	M	Secondary	N
Training sample 3	25	F	Primary	Y
Training sample 4	30	M	Primary	Y
Training sample 5	55	F	Secondary	N
New sample 1	*40*	*M*	*Primary*	*? → Y*
New sample 2	*30*	*F*	*Secondary*	*? → N*

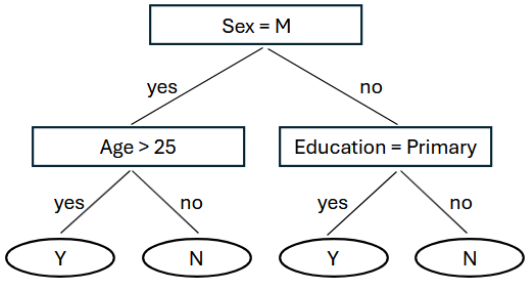


Figure 6.3: Illustration of a decision tree classifier.

In Figure 6.3 we provide an example of a decision tree. On the left the training set is given as well as the instances we try to predict. We have the age, sex and education of a person and want to find out if the person is married or not. On the right we can see a decision tree with three decision nodes and four leaf nodes for the prediction. For the new samples we can go through the decision tree to get the predictions. Starting from the root, when analyzing the new sample 1, we proceed to the left side, since the sex is M. Here, we further examine the age, which is greater than 25, which indicates marital status based on our decision tree. Similarly for new sample 2 we get the prediction that the person is not married.

6.1.3 Random Forest

One way to avoid the overfitting of the decision tree is to train multiple decision trees and use them together. This is what random forests do. After training several decision trees the new samples are given to each of the decision trees and the prediction of the

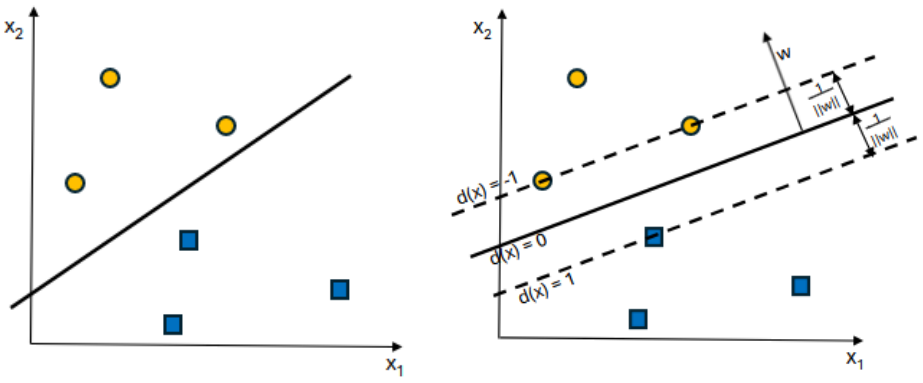
random forest is the majority of the predicted classes of the decision trees (classification) or the average of the predicted values of the decision trees (regression). This method is of course computationally more expensive and less interpretable. On the other hand, it not only reduces overfitting and handles noisy data well, it is also able to produce useful results for big datasets.

Such methods like the random forest approach, which combine several models are called ensemble learning techniques. Random forests were introduced by Ho, 1995.

6.1.4 Support Vector Machine

Support Vector Machines (SVMs) separate the classes by defining a hyperplane in the feature space, where the margin between the classes should be as big as possible. In this context the margin is the distance between the hyperplane and the nearest data point. One big advantage is that they cannot only do this with a linear decision boundary, but also using the "kernel trick", which is explained in the following. Each side of the decision boundary defines the space, where one class is predicted for new instances. Disadvantages of SVMs are that they are sensitive to parameter choices and the classical version of SVMs are only binary classifiers. SVMs were introduced by Boser et al., 1992.

We are going to explain how an SVM works using Figure 6.4 and perform classification with two classes starting with a linear decision boundary. On the left, in Figure 6.4a, we can see two classes, which are separated by a simple linear function. This is what a Perceptron does (a simple ML model). SVMs are based on this concept, but maximize the margin between the classes. This can be seen in Figure 6.4b, where the solid black line is the linear separation and the gap between the two dashed lines is the margin.



(a) Perceptron. (b) SVM.

Figure 6.4: Illustration of a perceptron and a support vector machine.

To define SVMs mathematically we introduce the labels -1 and 1 for our two classes. The training instances are called x_i with the corresponding true label t_i and the predicted

label of the SVM $d(x_i)$, where $i = 1, ..., N$ with N equal to the total number of instances. The first condition that has to be fulfilled is $d(x_i)t_i \geq 1$, since the true label and the predicted label have to be the same. Since our decision boundary is linear this can be written as $(w^T x_i + w_0) * t_i = w^T x_i t_i + w_0 t_i \geq 1$, where w is the coefficient vector and w_0 is the bias. The second condition is that the margin should be as big as possible. We define the margin as $\frac{2}{||w||}$ (see Figure 6.4b where the distance between the solid line and each dashed line is $\frac{1}{||w||}$). Maximizing the margin is equivalent to minimizing $||w||$ or minimizing $||w||^2 = w^T w$. The whole SVM can then be formulated as Lagrange problem:

$$L(w, \alpha) = \frac{1}{2} w^T w - \sum_{i=1}^{N} \alpha_i (w^T x_i t_i + w_0 t_i - 1) \tag{6.1}$$

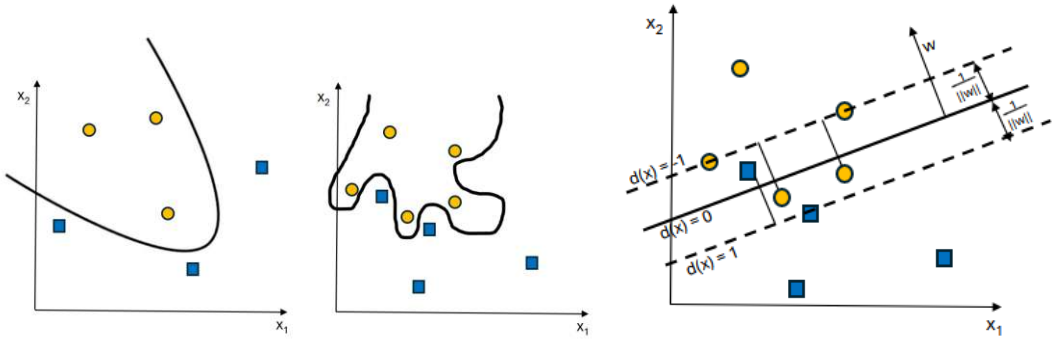
Since it is a Lagrange problem this primal version can be rewritten as dual formulation. To do that, we calculate the derivatives for our parameters w and w_0. This gives us the following conditions: $w = \sum_{i=1}^{N} \alpha_i t_i x_i^T$ and $\sum_{i=1}^{N} \alpha_i t_i = 0$. Substituting that into the primal formulation of the Lagrange problem and rearranging the terms gives us the dual:

$$L(\alpha) = -\frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j t_i t_j x_i^T x_j + \sum_{i=1}^{N} \alpha_i, \quad \alpha_i \geq 0 \tag{6.2}$$

Now the x vectors are only appearing in an inner product here. This is important since we now only looked at the linear case, but as mentioned above, the SVM is working for higher dimensional cases, too. This can be achieved by using different decision boundaries like illustrated in Figure 6.5a. Additionally, instead of having the problem of defining inner products in a higher dimensional space, we can replace the inner product by a kernel function like a sigmoid or polynomial function and since x only appears in inner products as we can see in Equation 6.2 it is sufficient to define a high dimensional kernel. This is called the "kernel trick". The corresponding dual Lagrange problem looks like the following, were $K(x_i, x_j)$ represents the kernel:

$$L(\alpha) = -\frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j t_i t_j K(x_i, x_j) + \sum_{i=1}^{N} \alpha_i, \quad \alpha_i \geq 0 \tag{6.3}$$

This gives us the possibility to achieve perfect separation even in very high-dimensional spaces. On the other hand, the behaviour in the plot in the middle of Figure 6.5 is not always the most desirable one, since these two classes could be linearly separated with only one misclassification. This means if we use the formulation of an SVM like in Equation 6.3 overfitting is very likely. Therefore, SVMs with soft margins are very useful because of the possibility to regularize the dimensionality of the decision boundary. What is the difference between a "classical" SVM and one with a soft margin? The SVM with a soft margin allows some instances to be on the wrong side of the decision boundary.



(a) SVMs with high-dim. decision boundaries. (b) SVM with constraints.

Figure 6.5: Illustration of support vector machines with kernel or constraints.

The misclassified instances still have to be within a provided ϵ range. As example we can see the SVM with a soft margin in Figure 6.5b. Be aware that the data points in the middle and right plot of Figure 6.5 are the same.

After formulating the SVM with a soft margin as Lagrange problem and reformulating it, we get the following dual formulation, which only has one additional constraint in comparison to the SVM dual formulation:

$$L(\alpha) = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j t_i t_j K(x_i, x_j) + \sum_{i=1}^{N}\alpha_i, \quad 0 \leq \alpha_i \leq \frac{C}{N} \tag{6.4}$$

In contrast to the introduced SVMs for classification it also works for regression. The goal has to be reformulated from maximizing the margin between the classes to minimizing the error between the true and predicted values, while still trying to keep the margin as big as possible. The concept is the same and again kernel functions can be used to represent higher dimensional relationships, as well as a soft margin formulation is possible. An overview of SVM regression is given by Smola and Schölkopf, 2004.

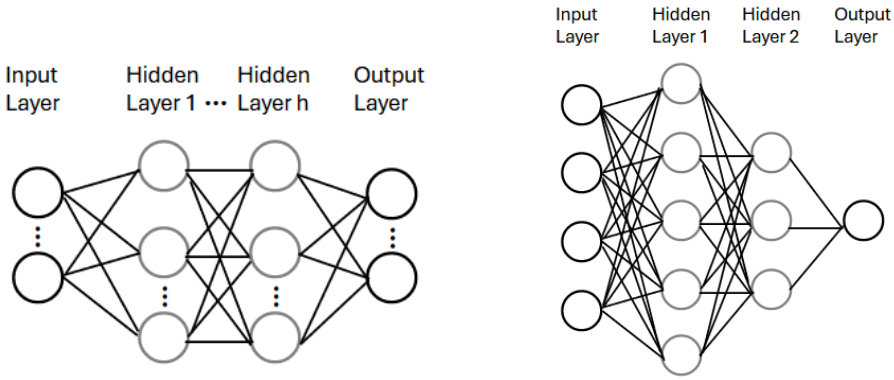
6.1.5 Multi-Layer Perceptron

Multi-layer perceptrons (MLPs) are highly flexible Neural Networks (NNs) and can be considered as deep learning models. This provides a lot of possibilities like deciding which architecture to use. Moreover, MLPs are good at handling high dimensional data and learning complex patterns. On the other side, they are often computationally expensive and less interpretable. An overview of MLPs and NNs in general is given in Popescu et al., 2009.

As the name suggests they are combinations of several perceptrons. One example of a perceptron is already shown in Figure 6.4a. As a function a perceptron looks like this: $y = w_0 + w^T x$. This formula represents the multiplication of the input vector with the weight matrix and the addition of the bias.

Now we want to be able to represent higher dimensional relationships. Therefore, we add activation functions. Examples of activation functions are the sigmoid function, tangens hyperbolicus (tanh) and the Rectified Linear Unit (RELU). We denote an activation function with σ. The new formula looks like this $y = \sigma(w_0 + w^T x)$. To get an MLP we additionally add at least one hidden layer. This means we multiply the input with weights and a bias and apply the chosen activation function on it and then we take these intermediate outputs and repeat the same procedure: multiply with weights, add the bias, apply the activation function. For one hidden layer the formula then is: $y = \sigma_2((w_0)_2 + w_2^T(\sigma_1((w_0)_1 + w_1^T x)))$. This can then be extended as far as wanted and needed. The activation function(s) and the number and size of the hidden layers can be chosen in any way. The training consists of giving inputs and outputs to the MLP and learning the weights in between. The procedure used for this process is called backpropagation, since it starts at the outputs and learns the weights back through the network to the input layer. This method is based on gradient descent and was described by Rumelhart and McClelland, 1987.

In Figure 6.6a we can see a schematic representation of MLPs with arbitrarily many hidden layers and chooseable sizes of the layers. In Figure 6.6b we can see one MLP architecture. It has two hidden layers with sizes 5 and 3. The input vector has size 4 and the prediction is a 1-dimensional output.



(a) Schema of MLPs. (b) Example MLP.

Figure 6.6: Illustration of fully-connected multi-layer perceptrons.

In this whole section we introduced the MLPs as fully-connected MLP, which means each unit of each layer has a connection with each unit of the next layer. This is not necessarily always the case. This additional flexibility allows for the selection of connections based on specific needs.

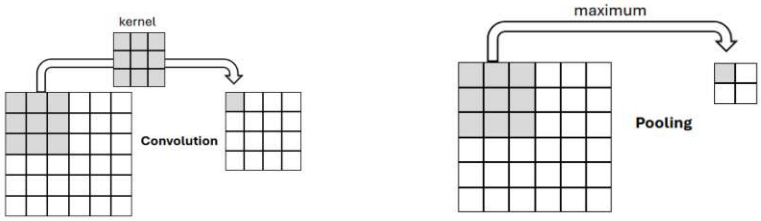
6.1.6 Hypergraph Neural Network

Neural Networks in general are highly flexible and as just mentioned for the MLP the number of layers and nodes, the activation function and a lot of other components can

be chosen based on different purposes. Until now, the inputs to the neural networks were numerical feature vectors. If a hypergraph is the basis of an ML problem, it can be a method to try to find features, which represent important characteristics of the hypergraph, but the structure of the hypergraph(s) contains so much information, that it would be nice to feed the whole hypergraph into the neural network, which learns the structure.

This is were Hypergraph Neural Networks (HGNNs) come into play. The idea is to represent the hypergraph somehow in a vector space and perform message passing through the neural network (Feng et al., 2019). If this is done, all possible layers can be stacked again to build a neural network on top of the hypergraph vector space.

We briefly want to introduce two common concepts used as neural layers: the convolutional layer and the pooling layer. Convolutional layers are kernels, which slide over the tensor of one layer and are multiplied with each part. The purpose is to learn patterns of the data by learning the weights of these kernels. In contrast to that, the purpose of pooling layers is to reduce the dimensionality by taking the average or maximum of some parts of the tensor. In Figure 6.7 both concepts are visualized. The functionality of MLPs, which is to multiply weights with the whole vector/tensor, is also called linear layer. Being aware of these different types of layers, one can be very creative in designing all kinds of neural networks.



(a) Convolutional layer. (b) Pooling layer.

Figure 6.7: Illustration of neural network layers.

6.1.7 Combination of MLP and HGNN

Knowing of all the tools explained in the last two subsections, it is easy to combine both variants with an additional neural network. This can be done in various ways and we will provide one possibility. The MLP can get (numerical) features as input, apply multiple linear layers on it and get an output vector v_{MLP}. In a similar way, the hypergraph can be put into the HGNN, which e.g. consists of several convolutional layers and a pooling layer, and leads to the output vector v_{HGNN}. Now these two vectors v_{MLP} and v_{HGNN} can be concatenated to one new numerical vector, which can be taken as input for another neural network. Again, all possible layers could be stacked for that one, e.g. some linear layers. In the end the output vector represents the input of the features and the hypergraph and the whole network learns to combine these information.

6.2 Data augmentation

Machine Learning models are always trained on data. Therefore, having enough training data is very often crucial for the performance and generalizability of the model. Nevertheless, for real world applications, it can be challenging to have/find a sufficient amount of data. In order to be able to use the desired models, data augmentation plays an important role. Data augmentation is the process of creating new training data based on the existing ones by changing something. There are a lot of existing data augmentation methods. Depending on the underlying data and data type they can vary a lot. Typical examples for tabular data is adding noise to some variables, shifting the data or oversampling the minority classes to handle class imbalances. For text data it is very common to replace letters or words randomly, with a probability or by synonyms. Data augmentation for visual data is very often used since images are very complex. Here, mirroring the image, changing contrast and brightness or adding random noise are typical examples. An overview of data augmentation techniques for different data types is given in Volkova, 2024.

6.3 Experiment design: Model selection

In this section we want to explain the experiment design of model selection, which is a framework for selecting the best performing model. An overview of the steps of such workflows is given in Raschka, 2020 and in Muller and Guido, 2018.

There are different ways to do this and several stages of this process can be performed in different ways. The experiment design, which we will use, is visualized in Figure 6.8.

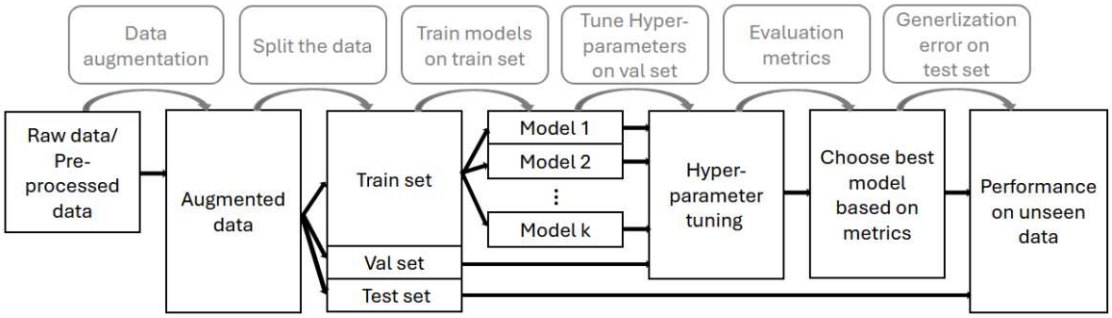


Figure 6.8: Experiment design.

The input to this workflow is the raw data or preprocessed data, which can be used for further steps. Preprocessing often consists of handling missing values or outliers depending on the available data. Then, data augmentation can be done if necessary, which we explained in Section 6.2.

One very important step is to split the data into a train and test set, where the test set is never used during the training process. This is important to be able to get a

meaningful and reliable result, since the data we want to apply to the model later on, is also not known in the training process. Very often an additional split is performed to get a train, validation and test set. In this case the validation set is used to compare the performance of the different models and the different hyperparameter settings on data, which the model (trained on the train set) did not use for the training. An alternative to the train-validation split is cross-validation, which is similar to several train-validation splits and inter changing the parts on which the training is done and the validation part. This is done to be able to use everything at least once for the training and meanwhile to ensure that the validation is always done on a set, which is not used for this training run. Again the cross-validation is done to compare the results of the different models and to compare the performance of different hyperparameters of the models, but the results are not that depend of the split, since each part of the dataset is used once for the validation. The best performing model on the validation set(s) is then used for the next steps. The test set is never touched during the whole process and only the chosen best model is applied on the test set in the end. Be aware that the terms validation and test set might be swapped in the literature sometimes.

As mentioned before several models are trained on the train set. The models are executed with different settings of the model parameters, which are often called hyperparameters (hyperparameter tuning). The model (with a set of hyperparameters) performing "best" on the validation set is chosen as "best" model. To figure out what "best" means, evaluation metrics are used to compare the different models and hyperparameter settings. We provide some of the most common evaluation metrics here.

For the validation set we have the true labels and we also get the predicted labels from one model. Starting with classification, we provide the confusion matrix for two labels ("positive" and "negative") in Table 6.1.

		Predicted	
		positive	negative
True	positive	TP	FN
	negative	FP	TN

Table 6.1: Confusion matrix.

If both labels are "positive" we get a true positive (TP), if both are "negative" we get a true negative (TN). Those are the cases we want, since it means our model classified the instance correctly. If the true label is "positive", but the model predicted "negative", we get a false negative (FN) and the other way around a false positive (FP).

Using these values we can define several metrics. The accuracy (acc) is defined as all correctly classified instances divided by all instances. The recall (rec) measures the proportion of the correctly classified positive classified instances of all actual positive ones, whereas the precision (prec) measures the proportion of the correctly classified positive classified instances of all predicted positive ones. Each of these metrics ranges between 0 and 1, where higher values are more desirable.

The formulas look like the following:

$$acc = \frac{TP + TN}{TP + TN + FP + FN}, \quad prec = \frac{TP}{TP + FP}, \quad rec = \frac{TP}{TP + FN}$$

It is advisable to look at the whole confusion matrix and each of these values, since one alone can often not fully express the behaviour of the model. For example a model classifying everything as positive, would achieve a recall of 100%. Furthermore, if recall or precision is more important depends on the problem and should be thought about, to choose the more suitable metric to evaluate the models for the given task. Be aware that the goal is to get high values for all those measures, but precision and recall have a trade-off relationship. An increase of one of them often leads to a decrease of the other one.

These concepts can of course be extended to multi-class classifications. For the accuracy the correctly classified instances for all classes are added and divided by the whole number of instances. For precision and recall there are two common methods for the multi-class case, which are micro-averaging and macro-averaging. Macro-averaging calculates the precision or recall for each class and then takes the average, whereas for micro-averaging the TP/FP/FN/TN over all classes are calculated and summed up first and then inserted in the precision or recall formula.

For regression tasks we have the true continuous values x_i and get the predicted ones y_i, where $i = 1, ..., N$, with N representing the total number of instances. One metric that can be used for regression is the Mean Absolute Error (MAE), which uses the average over all instances of the absolute difference between the true and predicted value. The Mean Squared Error (MSE) is very similar, but replaces the absolute differences by squared differences. This penalizes big differences between true and predicted value more. Additionally the Root Mean Square Error (RMSE) is often used, which, as the name tells us, takes the root of the MSE. For all of these metrics smaller values indicate a better model fit.

$$MAE = \frac{1}{n}\sum_{i=1}^{N}|x_i - y_i|, \quad MSE = \frac{1}{n}\sum_{i=1}^{N}(x_i - y_i)^2, \quad RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{N}(x_i - y_i)^2}$$

Finally, we want to mention the R^2-value, which represents how good the variance of the input variables is represented by the output and ranges between 0 and 1, where higher values indicate a better model fit. It is defined as the following

$$SSR = \sum_{i=1}^{N}(x_i - y_i)^2, \quad SST = \sum_{i=1}^{N}(y_i - \bar{y})^2, \quad R^2 = 1 - \frac{SSR}{SST}$$

This gives us the possibility to compare and choose between different models. Coming back to our experiment design workflow we now have the tools to choose the best performing model on the test set. This is the model we select for the application case.

To get an idea of how well that might perform on new unseen data, we now take our unused test set and apply this one model on it. The error (again measured with a suitable metric) between the true and predicted values of the test set is called generalization error. It should tell us how well it generalizes on unseen data and what performance we can expect in the future.

CHAPTER 7

Methodology

In this methodology chapter we describe the workflow of the thesis. An overview of this workflow is given in Figure 7.1. Each step is then described in more detail in the sections of this chapter. We start by describing the benchmark datasets we use including the

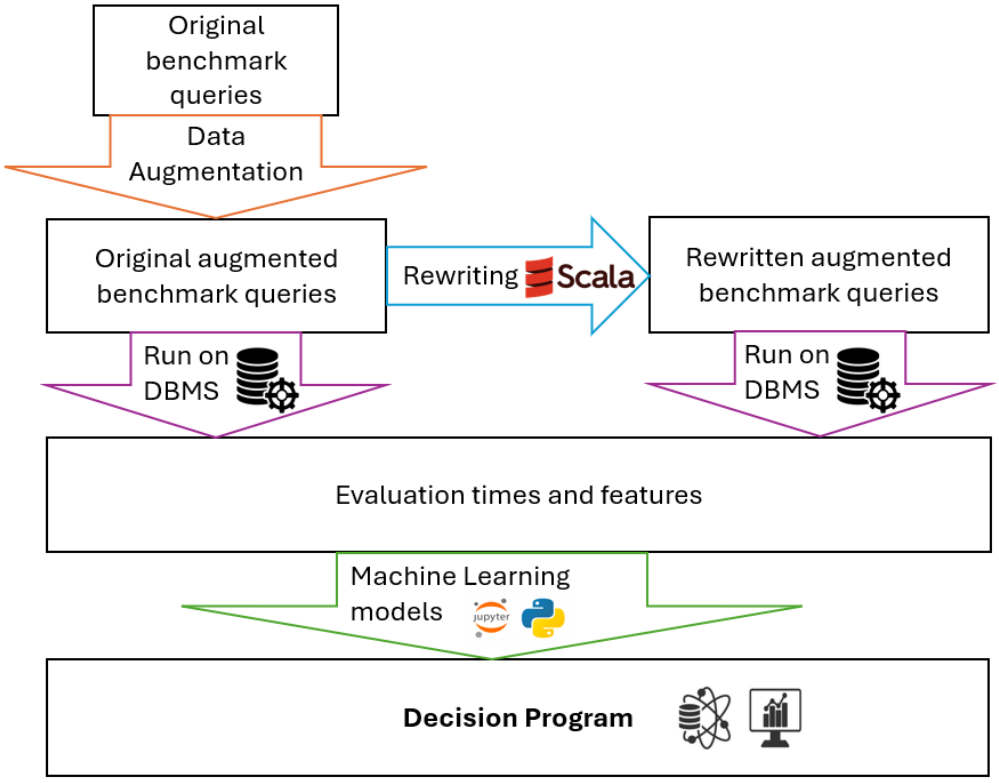


Figure 7.1: Methodology workflow.

data augmentation process in Section 7.1. Then, we explain the rewriting method and implementation in Section 7.2. Afterwards, we provide the DBMSs we use and some information about the query evaluation process in Section 7.3. Finally, in Section 7.4 we summarize how we get the decision program for the thesis' goal using ML models and describe the features we use.

The whole programming parts are provided in Github: https://github.com/danielaboehm/Query_optimization-Decision_program.

7.1 Benchmark Data Sets

We use several datasets to get data from different domains and designed for different tasks. We choose five benchmark datasets of different sizes and we explain them now briefly.

The STATS dataset introduced by Han et al., 2021 is designed to contain large scale joins on a join schema with several tables and different join forms as well as complex distributions and diverse workloads of real-world queries. The SNAP (Stanford Network Analysis Project) datasets are commonly used graph datasets. We decided to use four of them: cit-Patents, wiki-topcats, web-Google and com-DBLP (Leskovec and Krevl, 2014). The JOB (Join Order Benchmark) was designed by Leis et al., 2015 based on the real-world IMDB dataset. The purpose of this dataset is to test query optimizers. Therefore, it contains different numbers of joins and filter conditions to test the join ordering. The LSQB (Large-Scale Subgraph Query Benchmark) introduced by Mhedhbi et al., 2021 consists of nine queries based on an underlying graph and allowing to scale up to test query optimizers. The HETIONET dataset is based on a heterogeneous information network of biochemical data with the purpose of connecting multiple different sources into one real-world dataset introduced by Himmelstein et al., 2017.

In our thesis we only want to use CQs (for an explanation see Section 3.1). The queries in the benchmark dataset are CQs with additional filter conditions, but they can be considered to be CQs, since the filter conditions are only applied on single tables. This means, these filters are applied on the tables before joining, which leads to a CQ of the filtering step.

Some queries of the benchmark datasets are "SELECT MIN(...) FROM ..." queries and the others are "SELECT * FROM ...". For the latter ones we decide to transform them to "SELECT MIN(...) FROM ..." queries, in order to have 0MA queries (for an explanation see Section 5.5). The table we choose in the minimum is one random table occurring in the query and one column of this table (mostly the first column in the table). Soon we will see that it makes no huge difference which table we choose, as we exchange them anyway.

Additionally, we can use only acyclic queries, which we describe in Section 4.2 and this leads to dropping some of the queries from the benchmark datasets. The amount of (acyclic) queries of each dataset and the purpose of the datasets are given in Table 7.1.

Dataset	Purpose	# queries	# acyclic queries
STATS	real-world, all rel.	146	146
SNAP	graph queries	40	40
JOB	real-world, foreign key rel.	33	5
LSQB	test optimizers	9	2
HETIONET	real-world, graph queries	26	26

Table 7.1: Benchmark datasets.

Overall, we have 219 acyclic queries. Since we want to train an ML model this is not enough training data. Therefore, we do data augmentation. The general concept of data augmentation is explained in Section 6.2. For our data we decided to use two self-designed steps for data augmentation: the "filter augmentation" and the "aggregate-attribute augmentation". This can be seen in Figure 7.2.

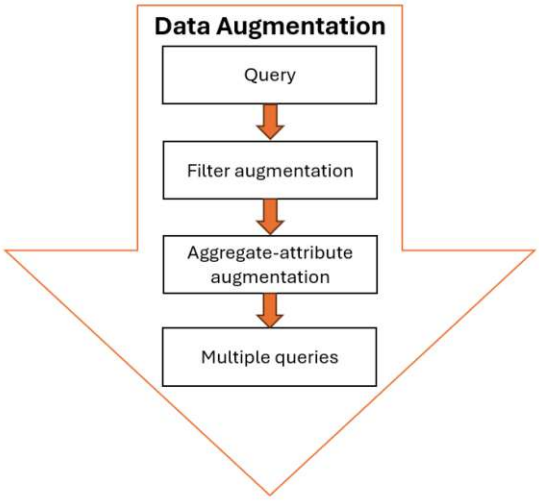


Figure 7.2: Steps of data augmentation.

With the filter augmentation we want to get duplicates of all queries having filters and then change some filters in a way that the sizes of the tables vary between these queries. If the query had only one filter we change the specific value it is equal to, greater or smaller of the filter condition. For these cases we get twice as many queries as before. For all queries having two or more filters we choose two filters, which we change each at a time. This is done by hand and we try to replace the filters in a way that once the number of answer tuples gets bigger and once smaller. This gives us triples for each of these queries. As an example the STATS query 005-024 could be filter-augmented in the following way, where the first one is the "original" query and the other two are the augmented ones. In this case, once the filter condition "v.BountyAmount>=0" is transformed to "v.BountyAmount>=40" and the other time "u.DownVotes=0" to "u.DownVotes=10".

```
005-024: SELECT MIN(v.Id)
         FROM votes as v, badges as b, users as u
         WHERE u.Id = v.UserId AND v.UserId = b.UserId
               AND v.BountyAmount>=0 AND v.BountyAmount<=50
               AND u.DownVotes=0
005-024-augF1: SELECT MIN(v.Id)
               FROM votes as v, badges as b, users as u
               WHERE u.Id = v.UserId AND v.UserId = b.UserId
                     AND v.BountyAmount>=40 AND v.BountyAmount<=50
                     AND u.DownVotes=0
005-024-augF2: SELECT MIN(v.Id)
               FROM votes as v, badges as b, users as u
               WHERE u.Id = v.UserId AND v.UserId = b.UserId
                     AND v.BountyAmount>=0 AND v.BountyAmount<=50
                     AND u.DownVotes=10
```

The "aggregate-attribute augmentation" is to exhaustively exchange the table we give into the MIN. This is done in a way that each table, which appears in the query, appears once in the MIN. The column of the chosen table is random, which means we just take the first column of the table. Depending on the number of tables involved in the query this leads to a different number of new queries per query. For the example STATS query this can look like the following, where the three tables are each represented once in the minimum.

```
005-024: SELECT MIN(v.Id)
         FROM votes as v, badges as b, users as u
         WHERE u.Id = v.UserId AND v.UserId = b.UserId
               AND v.BountyAmount>=0 AND v.BountyAmount<=50
               AND u.DownVotes=0
005-024-augA1: SELECT MIN(b.Id)
               FROM votes as v, badges as b, users as u
               WHERE u.Id = v.UserId AND v.UserId = b.UserId
                     AND v.BountyAmount>=0 AND v.BountyAmount<=50
                     AND u.DownVotes=0
005-024-augA2: SELECT MIN(u.Id)
               FROM votes as v, badges as b, users as u
               WHERE u.Id = v.UserId AND v.UserId = b.UserId
                     AND v.BountyAmount>=0 AND v.BountyAmount<=50
                     AND u.DownVotes=0
```

Be aware that all filter augmented queries also get aggregate-attribute augmented. For example for this query we get 9 queries after the whole augmentation.

In Table 7.2 we summarize the queries we have got after each step of the augmentation. The SNAP and LSQB queries do not have filter conditions, which means there is no filter-augmentation for them. Overall, we get 2936 queries, which we use in the next steps of the workflow.

Dataset	# acyclic queries	after filter aug.	after filter + agg aug.
STATS	146	432	1876
SNAP	40	40	244
JOB	5	45	264
LSQB	2	2	14
HETIONET	26	72	538

Table 7.2: Number of queries with augmentation.

7.2 Rewriting method and implementation

The rewriting method is the process of taking one query and getting a sequence of new SQL queries, which give the same result, but force the DBMS to evaluate the query with a method based on Yannakakis' algorithm. We take the existing rewriting method of Gottlob et al., 2023 and Spark code from Lanzinger et al., 2024, which is given at https://github.com/arselzer/spark/blob/v3/sql/catalyst/src/main/scala/org/apache/spark/sql/catalyst/optimizer/RewriteJoinsAsSemijoins.scala and implement it in Scala to add the possibility of using filter conditions and aggregates in the SQL statement.

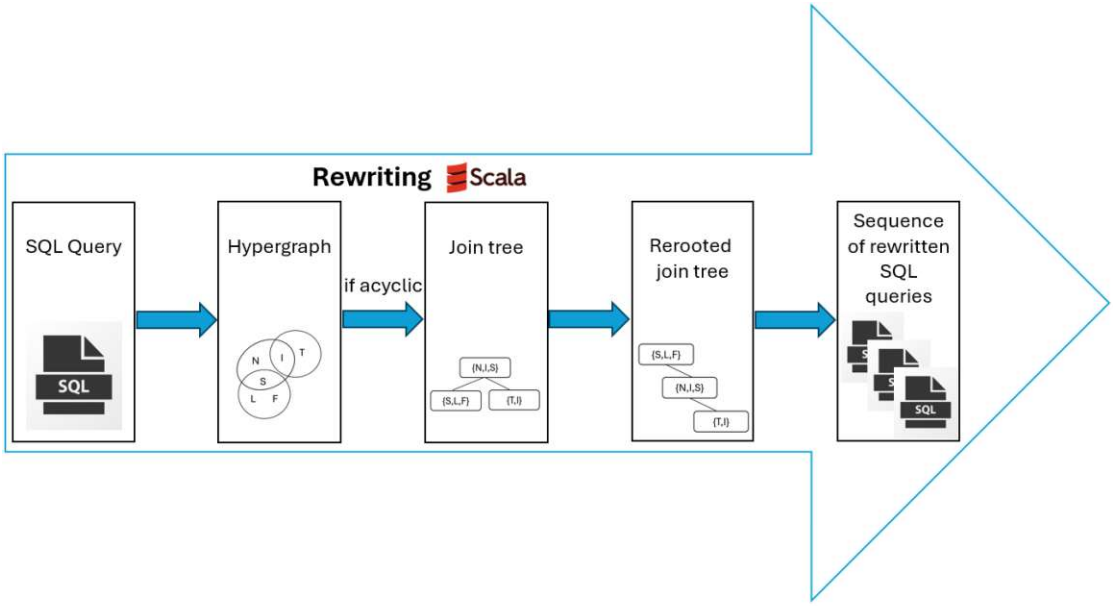


Figure 7.3: Scala implementation for rewriting the queries.

As visualized in Figure 7.3 the SQL query is first transformed into a hypergraph, which we described in Section 4.1. With the GYO-reduction we get a join tree if the query is acyclic. This procedure is explained in Section 4.3. If the query is not acyclic, our program would stop and print that the query is cyclic. In case that the query is acyclic, which is the case for our used 219 queries (see previous chapter Table 7.1), we get a join tree from the GYO-reduction. The aggregate always consists of a column of one table and this table now should be the root to give us the possibility to evaluate the query with only one traversal of the join tree. This is possible since we have 0MA queries (for further explanation see Section 5.5). After this "rerooting" we get a join tree, which gives us the desired join order and since a join tree can be seen as a query execution plan (Section 4.2), we can use it to create the new sequence of SQL query. This is done from the leaf nodes of the join tree up the tree by creating a "CREATE VIEW ..." or "CREATE TABLE ..." command for each of the nodes of the join tree. At the root node the aggregation is added and the last command is to select everything from the root node to get the result. For the STATS query 005-024-augA2, which we already used as example in the last section, the query before and the sequence of queries after the rewriting looks like the following:

```
005-024-aug2: SELECT MIN(u.Id)
              FROM votes as v, badges as b, users as u
              WHERE u.Id = v.UserId AND v.UserId = b.UserId
                    AND v.BountyAmount>=0 AND v.BountyAmount<=50
                    AND u.DownVotes=0

005-024-rewr1: CREATE VIEW E3 AS SELECT *
               FROM users AS users
               WHERE users.DownVotes = 0
005-024-rewr2: CREATE VIEW E2 AS SELECT *
               FROM badges AS badges
005-024-rewr3: CREATE UNLOGGED TABLE E3E2 AS SELECT *
               FROM E3 WHERE EXISTS (SELECT 1
                                     FROM E2
                                     WHERE E3.Id=E2.UserId)
005-024-rewr4: CREATE VIEW E1 AS SELECT *
               FROM votes AS votes
               WHERE CAST(votes.BountyAmount AS INTEGER) >= 0
                    AND CAST(votes.BountyAmount AS INTEGER) <= 50
005-024-rewr5: CREATE UNLOGGED TABLE E3E2E1 AS
               SELECT MIN(Id) AS EXPR$0
               FROM E3E2 WHERE EXISTS (SELECT 1
                                       FROM E1
                                       WHERE E3E2.Id=E1.UserId)
005-024-rewr6: SELECT * FROM E3E2E1
```

Be aware that the table, which is contained in the minimum, is sometimes not the root node. This is possible if the attribute in the minimum is one, which is part of a equi-join. Then, the other table of the join can be the root, since then same attribute is still in the minimum, just with another table. For our example the rerooted join tree could also have votes or badges as root, since the minimum is on users.id, but this is equal to votes.UserId and badges.UserId.

Moreover, in the rewriting of the example query, we can see that users and badges are joined first, even if they did not have an equality condition together in the original query. This is again due to the fact that the three tables are joined on the same attribute.

In the implementation we additionally add a function, which gives us a sequence of DROP statements to be able to delete the created tables of the rewriting after the evaluation again.

Furthermore, some features for the ML models we use later for the decision program are based on the join tree and calculated and saved in the Scala program. The features are described in more detail in Section 7.3.

7.3 DBMSs

Now we can evaluate the query as "original" or "rewritten" version. The evaluation is done with three different DBMSs: PostgreSQL, DuckDB and SparkSQL. PostgreSQL is a well-established relational DBMS, whereas DuckDB is an in-process, column-oriented DBMS and SparkSQL is part of a distributed cluster environment. We decide to use these three DBMSs since they have different architectures and therefore give a good overview of the range of existing DBMSs.

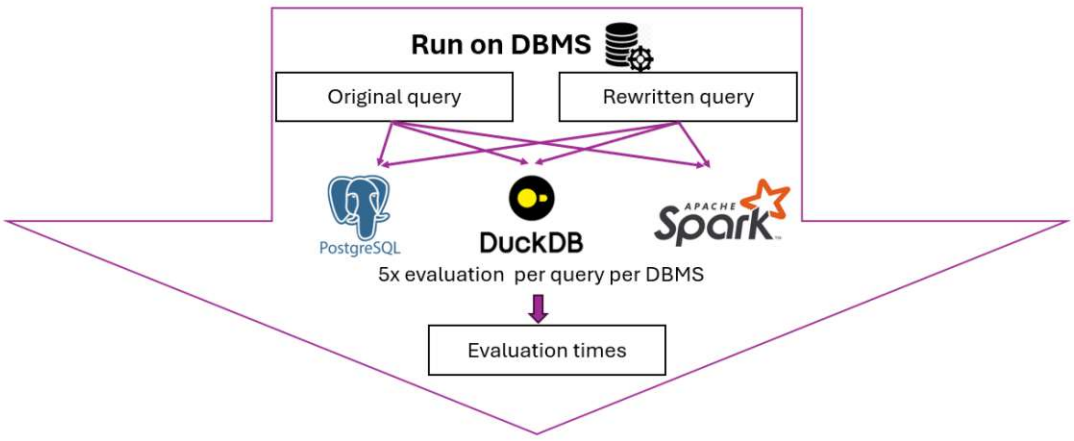


Figure 7.4: Workflow of the query evaluation.

In Figure 7.4 we illustrate this part of the workflow. Each query is once evaluated without taking the time. Then, the query is evaluated five times with taking the time for each

of the evaluations with both variants (original and rewritten query). The first run is considered as a warm-up, so that the time to load a table into memory for the first time does not affect the results.
The whole evaluation is done on a server with 128GB RAM, 16 virtual CPUs and 100GB disk storage.

7.4 Decision program with ML models

To get the decision programs using ML models, we follow the experiment design for model selection provided in Section 6.3. The part with the specific ML models and their input and output are visualized in Figure 7.5.

We get several decision models for each of the three DBMSs, depending on different features and settings. We compare them in Section 8. In this section we provide the general approach to get one decision program and describe all features we use in at least one of the ML models.

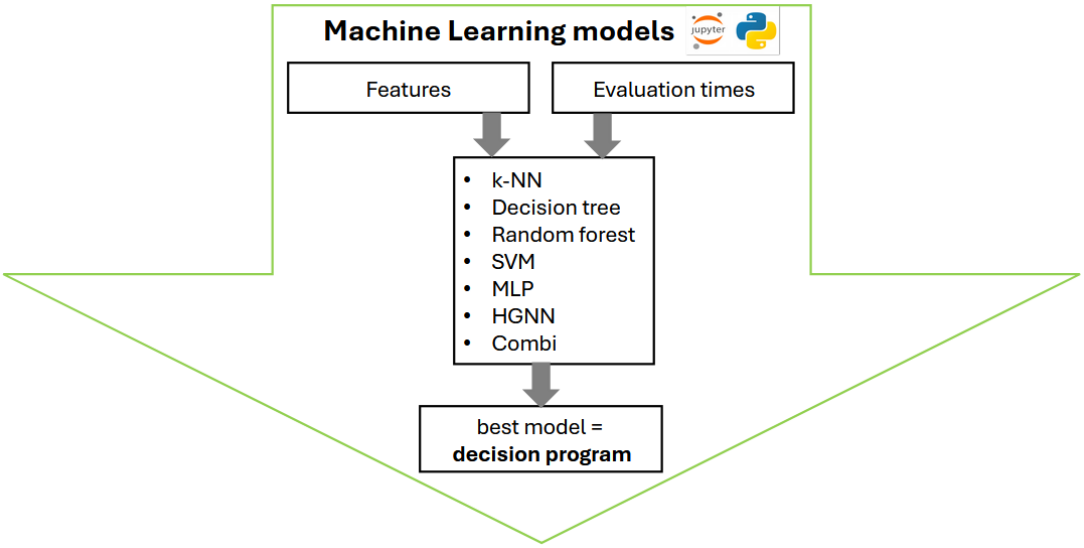


Figure 7.5: Inputs and output of the used ML models.

As described in the experiment design, we split the data into train, validation and test set. The train set contains 80% of the data and the other two 10% each. The splitting is done with stratification, which means each of our five benchmark datasets is represented in each of the three sets approximately with the same proportion.

The data we have consists of the queries and their evaluation times. Now we create features, which we then put into the ML models as inputs, together with the evaluation times as response values. There are different kinds of features we would like to use. We

have features based on the query structure, we use features based on the join tree and features, which we get from the databases PostgreSQL or DuckDB.

7.4.1 Features

We start by describing the features, which we get for each query just by "looking" at the query itself.
Number of relations: This feature is representing the number of all relations part of the query.
Number of conditions: For this feature the number of (in)equality conditions in the WHERE clause of the CQ are counted.
Number of filters: Here, only the (in)equality conditions, which are filters, are counted.
Number of joins: For this feature the number of joins, which can also be computed by looking at the equality conditions, are calculated. Be aware that sometimes two equality conditions belong to one join.

Next, we provide some features based on the join tree, which we get for each of our (acyclic) queries with the GYO-reduction in the Scala programming part. These features are inspired by Abseher et al., 2017.
Depth: For this feature the maximal distance between the root of the used join tree and a leaf node is computed.
Container counts: Here for each variable the number of nodes it occurs in is counted. Since we get a value for each variable we can calculate several statistics, which are the minimum, maximum, mean, median, first and third quartile. This measure indicates how many relations are joined on the same variable.
Branching factors: For this feature we count the number of children for each node. This gives us different numbers again and we use the same statistics as above.
Another feature using the structure of the join tree is the balancedness factor described in Selzer, 2021. This feature should represent how balanced the tree is. If each node has the same amount of children, the tree is perfectly balanced. If we have a left-deep tree or a right-deep tree, the tree is not balanced. To calculate this feature, we count the sizes of the subtrees of each node and divide the minimum by the maximum. The balancedness factor is the average of all those numbers. This gives interesting information if parallelization is taken into consideration, but since we do not do this, we drop this feature.

We call this first seven features the "basic features", since we get them for every query and every DBMS. We can also use information from the databases. First, we describe the feature, which we get by the EXPLAIN command in PostgreSQL. This uses the query and provides the execution plan with some cost estimations, which we can use as features. We refer to them as "POS features". (This is not an evaluation yet.)
Estimated total cost: The first feature is to take the estimated total cost.
Estimated single table rows: Then, we count the estimated number of rows for each table involved in the query. This gives us the estimated row numbers after the filters are applied on the table. This again gives us multiple values and as features we use the

statistics, like the mean or median as mentioned above.
Estimated join rows: To gain some information about the intermediate results, which often are the reason for long execution times, we also take the estimated number of rows of each join into consideration. This gives us a value for each join operation part of the execution plan, where we again can use the statistics to get our features.
Foreign key/primary key relations are indirectly represented in the estimated rows, since PostgreSQL considers that for planning.

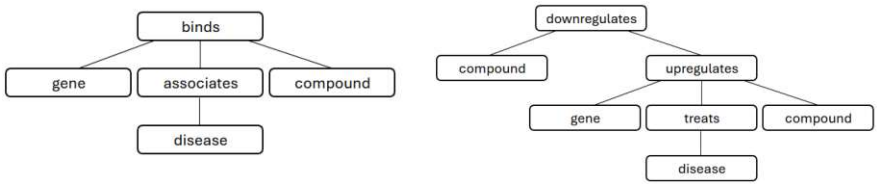
We get similar features for DuckDB using the EXPLAIN command for DuckDB. This again gives information about the logical execution plan, but only provides the estimated cardinality. We call these the "DDB features".
Estimated cardinalities: This feature gives us the estimated cardinality, which is the number of rows after reading a table, filtering and joining tables.

SparkSQL does not provide such a command like EXPLAIN, which means that we do not get additional features about the execution plan or other database information of SparkSQL. For better illustration of the features, we give two examples together with all of the features: Q1: HETIO_2-01-CbGaD, Q2: HETIO_3-06-CdGuCtD.

```
Q1: SELECT MIN(c.nid)
    FROM compound c, binds b, gene g, associates a, disease d
    WHERE c.nid = b.sid AND b.tid = g.nid
          AND g.nid = a.tid AND a.sid = d.nid
```

```
Q2: SELECT MIN(c1.nid)
    FROM compound c1, downregulates d1, gene g, upregulates u2,
         compound c2, treats t, disease d
    WHERE c1.nid = d1.sid AND d1.tid = g.nid
          AND g.nid = u2.tid AND u2.sid = c2.nid
          AND c2.nid = t.sid AND t.tid = d.nid
```

The corresponding join trees calculated by Scala are given in Figure 7.6. They help us to identify some of the features more easily.



(a) Joint ree of Q1. (b) Join tree of Q2.

Figure 7.6: Calculated join trees of the example queries.

	Q1	**Q2**
#relations	5	7
#conditions	4	6
#filter	0	0
#join	4	6
depth	2	3
container counts	*1, 1, 1, 1, 1, 2, 3*	*1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 3*
min(container counts)	1	1
max(container counts)	3	3
mean(container counts)	1.43	1.27
q25(container counts)	1	1
median(container counts)	1	1
q75(container counts)	1.5	1
branching factors	*3, 1*	*2, 3, 1*
min(branching factors)	1	1
max(branching factors)	3	3
mean(branching factors)	2	2
q25(branching factors)	1.5	1.5
median(branching factors)	2	2
q75(branching factors)	2.5	2.5
POS: total cost	1175.3	10283.64
POS: table rows	*23142, 25246, 137, 1, 1552*	*154076, 1552, 146276, 1510, 137, 1552, 20945*
POS: min(table rows)	1	137
POS: max(table rows)	25246	154076
POS: mean(table rows)	10015.6	46578.29
POS: q25(table rows)	137	1531
POS: median(table rows)	1552	1552
POS: q75(table rows)	23142	83610.5
POS: join rows	*1190, 2361, 626, 626*	*493338, 22993, 2578, 2578, 446, 446*
POS: min(join rows)	626	446
POS: max(join rows)	2361	493338
POS: mean(join rows)	1200.75	87063.17
POS: q25(join rows)	626	979
POS: median(join rows)	908	2578
POS: q75(join rows)	1482.75	17889.25
DDB: cardinality	*31829, 29051, 26334, 26334, 418, 2299, 29051, 24035, 26334*	*4993749, 354275, 338789, 322069, 2299, 2299, 338789, 2090, 24035, 2299, 322069, 418, 2090*
DDB: min(cardinality)	418	418
DDB: max(cardinality)	31829	4993749
DDB: mean(cardinality)	21742.78	515790
DDB: q25(cardinality)	24035	2299
DDB: median(cardinality)	26334	24035
DDB: q75(cardinality)	29051	338789

Table 7.3: Features.

In Table 7.3 all features of these two queries are given. Additionally, we provide all values of the container counts, branching factor, table rows, join rows and cardinality in italic to give an insight into how these features work. Be aware that these features are never used all together for one model. The model construction and which features are used, is explained in the next section.

7.4.2 Models and Evaluation

In this section, we describe which models and features we use for our experiments. In general, we use some of the features described in the previous section as input for our models and we want to predict, if the evaluation of the original query or the rewritten query is faster. For our training data, we got the runtimes for each of the queries with each of the DBMSs. The training label then can be used as classification, where it represents if the original or the rewritten was faster (0,1 encoded), or as regression, where the time difference between the two versions can be used. We also try another classification, where we introduce a third class "equal", where all queries are contained, which have almost equal runtimes. For this purpose we choose a cut-off value and if the time difference between the queries is below this threshold, the label is changed to equal, no matter if the original or rewritten version was a little bit faster. This is done for four cut-offs: 0.5, 0.1, 0.05 and 0.01.

First, we apply different ML models for each of the three DBMSs using the basic features. We use the seven models described in Section 6.1: k-NN, decision tree, random forest, SVM, MLP, HGNN and a combination of MLP+HGNN. This gives us a very good opportunity to compare the performance of the models for the three DBMSs. Then, we try to get more accurate results using additional features. We use the basic features combined with the PostgreSQL features and the PostgreSQL runtimes. Similarly, we use the basic features and the DuckDB features together with the runtimes of DuckDB. Since SparkSQL does not provide such additional features, we decided to use the basic features together with the POS features, to see if that improves the results. We are interested in comparing these models with those using only the basic features. For all these cases we apply all the seven ML models again.

In Table 7.4 we summarize all hyperparameters used for each model. We decide to take a fixed number of five neighbors for the k-NN and the number of combined decision trees as 100 for the random forest. For the SVM we use three different kernels.

For the deep learning models we have to introduce additional parameters, which we did not vary. As loss function, which measures the prediction error, we use the MSE (for an explanation see Section 6.3) for the regression tasks and the Cross Entropy for the classification tasks. The Cross Entropy measures the difference between the predicted and actual probability distribution of each class. The epochs state how often the entire dataset is used in the training process. All deep learning models want to learn weights and the learning rate is the step size of the weight update. This means the current calculated weight of one epoch is multiplied by the learning rate and then added to the

model	hyperparameter	layer
k-NN	k=5	
Decision tree		
Random forest	n_estimators=100	
SVM	kernel=linear/poly/rbf	
MLP	Loss=Cross-Entropy/MSE Batch size=100 Epochs=300 (saving best model) Leaning rate=0.1	in-5-out
		in-10-out
		in-20-out
		in-25-out
		in-40-out
		in-60-out
		in-10-5-out
		in-20-10-out
		in-40-20-out
		in-40-10-out
		in-60-40-out
		in-60-20-out
		in-80-50-out
		small median, best MLP
		small mean, best MLP
		small min, best MLP
		small max, best MLP
		small q25, best MLP
		small q75, best MLP
		custom, best MLP
HGNN	Loss=Cross-Entropy/MSE Epochs=100 (saving best model) Leaning rate=0.001 Max-Pooling	kernel 3x3, 1-16-32-out
		kernel 3x3, 1-32-16-out
		kernel 3x3, 1-16-32-16-out
		kernel 3x3, 1-32-64-out
		kernel 3x3, 1-4-16-out
combined	Loss=Cross-Entropy/MSE Epochs=100 (saving best model) Leaning rate=0.001 Max-Pooling	best MLP-2/best HGNN-2/4-out
		best MLP-5/best HGNN-5/10-out
		best MLP-5/best HGNN-5/10-20-out
		best MLP-10/best HGNN-10/20-40-2
		best MLP-10/best HGNN-10/20-60-20-2

in = number of features, out = number of classes (2,3 and 1 for regression)
Loss: Cross-Entropy for classification, MSE for regression

Table 7.4: Hyperparameters.

old weight to get the new weight. Since the training sample size can be big it is often advantageous to use smaller parts of the dataset for an update. The size of these parts is given by the batch size. So, if there are 1000 training samples and the batch size is 100, then there are 10 batches per epoch, which means 10 weight updates per epoch. For the HGNN we use the max-pooling as pooling layer, which is explain in Section 6.1.6.

The number of hidden layers for the MLP is either one or two, where the number of nodes is varied and provided in the table. Additionally, the number of features is reduced, since several features occur multiple times as min, max, mean, median 25%- and 75% quartiles to just one of these statistics. Another version is to add a custom layer, which summarizes all six statistical values for one feature to one node. The HGNN has two or three convolutional layers with a kernel size of 3x3 and then one max-pooling layer. For the combination of the MLP and HGNN we use the best performing model of the MLPs and the best performing model of the HGNNs and combine the outputs, where one, two or three linear layers are applied on.

The results of all these models including the comparisons are given in Section 8, as well as the resulting decision program(s).

Moreover, for the k-NN, decision tree, random forest and SVM cross validations are done additionally. This means the 80% of the data, which are the training set and the 10% of the validation set are put together and then used for 10-fold cross validation.

To be able to compare the models and decide, which are the best ones, we need to use metrics as defined in Section 6.3. We use the accuracy for the classification and the MSE for the regression version. Since one should not only use one quantitative metric, we also provide an inspection of the model and the misclassified instances. In Section 8 we will get the results and choose the best three models overall. For those models we provide additional metrics like precision and recall and do qualitative analysis of the misclassifications.

Finally, we will get one "best" model, which we then apply on the untouched test set. For that we do quantitative analysis (metrics) and qualitative analysis (observe the misclassifications) similarly as before. Additionally, we perform two statistical tests to be able to conclude if we can achieve significantly better results. The tests compare the mean or median of the runtimes for the following two cases: 1. runtimes of the queries of the test set with the original version and 2. runtimes of the queries of the test set with either the original or the rewritten version, based on the decision we made by our decision program.

Since both of these cases use the data in the test set, we need to use statistical tests, which take this dependencies into consideration. For the median we take the Wilcoxon sign-rank test (Wilcoxon, 1945) and for the mean we use a paired sample t-test.
Wilcoxon sign-rank test: The null hypothesis of this test for two (dependent) groups A and B is that the medians are equal: $H_0 : median(A) = median(B)$. To get the test statistic the differences between all pairs of group A and B are calculated and ranked. Additionally, the sign of the difference is used, so that all ranks of the positive differences

are summed and the same for the negative ones. The minimum of these two is the test statistic, which then can be compared to the Wilcoxon signed rank table to get the p-value. If the p-value is smaller than a chosen alpha-level, the null can be rejected and the two cases lead to significantly different medians.

Paired sample t-test: The null hypothesis of this test for two (dependent) groups A and B is that the means are equal: $H_0 : mean(A) = mean(B)$. Again, the differences of the pairs of values are used to calculate the test statistic. In this case it is a t-test statistic with n-1 degrees of freedom and looks like the following.

$$t = \frac{\bar{d}\sqrt{n}}{s}, \qquad \text{with } \bar{d} = \frac{1}{n}\sum_{i=1}^{n} d_i, s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(d_i - \bar{d})^2}$$

Here the t-test tables can be used to get the p-value and again if it is smaller than alpha, the null can be rejected and we can conclude that the means are significantly different.

CHAPTER 8

Results

This chapter contains all results and insights of this thesis. For each of the three DBMSs, PostgreSQL, DuckDB and SparkSQL, general characteristics of the runtimes of the original and rewritten queries run on them like the distribution, the order of magnitudes and how the datasets influence the runtimes are presented. After that all ML models applied on that data with the basic features are inspected based on their metrics and how well they work as decision programs (for the workflow see Section 7.4). This means the regression, too, is used for predicting the label based on a cut-off value. Additional to that quantitative analysis, we perform qualitative analysis in a way that we observe the misclassifications for the most promising models in more detail. In particular we look at how big the time difference between the runtime of the original query and the runtime of the rewritten query is, if the label is predicted wrongly. If the time difference between the two versions is very small, which means that the runtimes are almost the same, then it is not a big problem that the instance was misclassified, since it makes no big difference. The same procedure is applied on the data for each DBMS, in Section 8.1 for PostgreSQL, in Section 8.2 for DuckDB and in Section 8.3 for SparkSQL, where then additional features are added and the impact of adding them is observed.

After inspecting which results can be achieved for each DBMS, we want to find out, which ML model suits best for the task. Therefore, the top three models are listed for each DBMS and classification or regression. With that and the analysis of the results before, we can decide which model is the "best" performing model for our purpose. This is done in Section 8.4 and then, this "best" model can be applied on the final test set in Section 8.5. The test set was untouched before and helps us to find out how well our final model generalizes on unseen data. This is again observed with quantitative (metrics) and qualitative (misclassifications) analysis. Moreover, statistical tests are performed to find out if the decision method outperforms the original version in terms of mean and median runtimes. Finally, the final "best" model is visualized and inspected as well as the most important features are detected.

In the first section we will explain some important aspects in more detail and refer to them in the later sections, since they occur multiple times. Additionally, only the most important numbers are provided in the tables of this chapter, but all observed results are listed in the Appendix A.

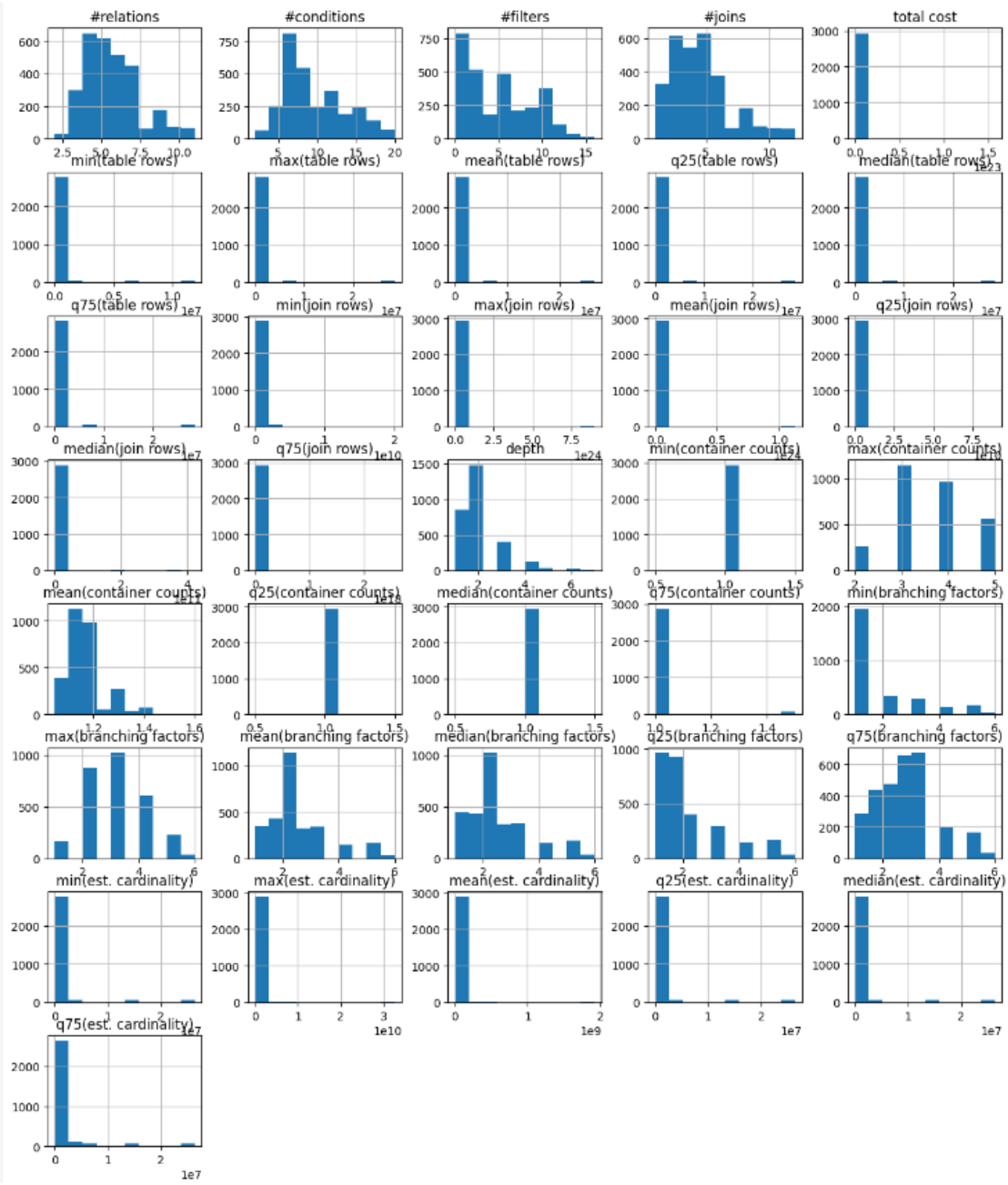


Figure 8.1: Distributions of the features.

Before showing the results of the ML models, we briefly have to talk about the input. We show the distributions of all features used in at least one model in Figure 8.1. Some of the input features are highly skewed and have to be log-transformed to obtain more reliable results.

The features which need a transformation are "total cost" and the minimum, maximum, mean, median, 25%-quantile and 75%-quantile of "join rows", "table rows" and "cardinality". The transformation leads to a distribution closer to a normal distribution, which is what we try to achieve for the input values of ML models, such that outliers do not have too much influence. The transformed features are visualized in Figure 8.2.

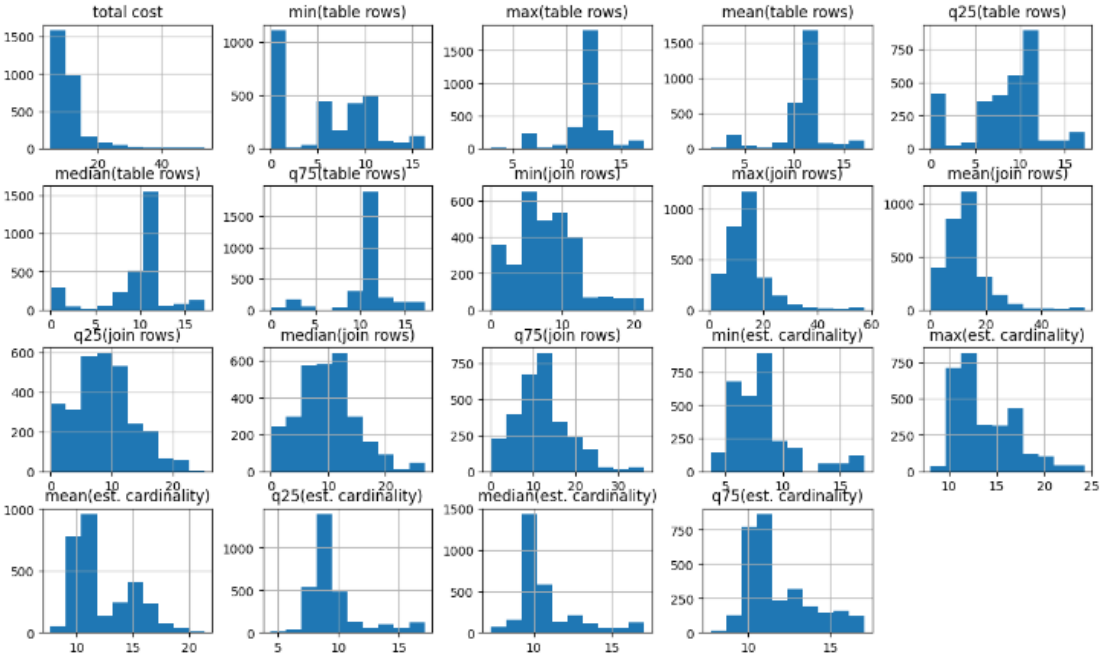


Figure 8.2: Distributions of the log-transformed features.

The cross validation results are not inspected in detail in this chapter, since not all models have a cross-validation version and additionally the results are very similar to the train-val split results, but all values are given in Section A.7 in the Appendix.

8.1 PostgreSQL

The first DBMS, which we have used to get the runtimes of the original and rewritten queries, is PostgreSQL. First, we take a look at the distribution of these runtimes split up in orders of magnitude. This is done in Figure 8.3, where the original runtimes are given on the left and the rewritten on the right. Additionally, the portions of each dataset per order of magnitude are colored.

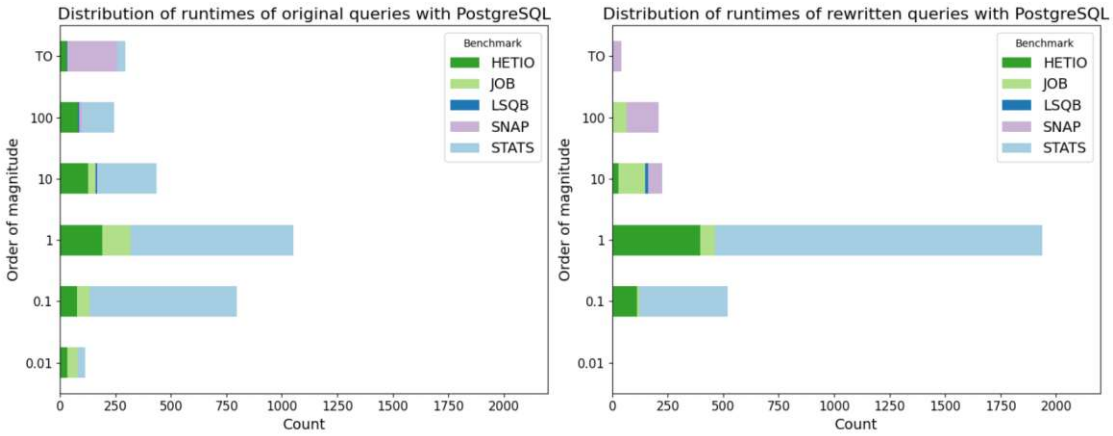


Figure 8.3: Comparison of distributions of orders of magnitude for PostgreSQL.

We can see that there are a lot more timeouts for the original queries, especially for SNAP queries, than for the rewritten ones. On the other hand, there are also more queries in the original version, which are executed really fast (runtime smaller than 0.1). This is exactly the expected behaviour, where neither version is always better than the other one, and our decision program should help to use the better version for each query. The SNAP and LSQB queries tend to have longer runtimes, whereas the queries of the other three datasets often are executed very fast.

Now we partition the runtimes into classes. Either the original or the rewritten version is faster, which gives us the two classes for the 2-class classification. As explained above we also introduce thresholds if the runtimes are very similar to be considered as "equal" classes. The amount of queries in each class for each version is provided in Table 8.1.

	2 classes	3 classes (0.01)	3 classes (0.05)	3 classes (0.1)	3 classes (0.5)
orig	1418	1322	1007	706	240
rewr	1480	1435	1336	1243	989
equal	-	141	555	949	1669

Table 8.1: Distribution of the classes for the classifications for PostgreSQL.

For the 2-class case the number of queries in both classes is very balanced. Depending on the threshold the amount of queries in the three classes differ. We observe that more of the queries in the equal class, which are those where the runtimes are nearly the same, are "orig" queries. This tells us, if the rewriting is faster it improves the runtime a lot and if the original is faster, it is often just a little bit faster than the rewriting.

After considering the classification response, we also want to inspect the regression response. The regression response is the time difference between the two versions, where the original runtime is subtracted from the rewritten runtime. In Figure 8.4 on the left we can see the distribution of the time differences. The time difference has a wide

range and is a bit skewed. Therefore, we are going to transform it. As before with the features, we would like to apply a log transformation. Nevertheless, since we have negative values this cannot be applied directly. We are going to multiple the log of the absolute values with the sign they had before. Additionally, since we have a lot of values close to zero, which leads to very small log values, we add 1 to the absolute values before applying the log, which is a common method. The transformation as formula looks like the following: $x_{new} = sgn(x) * log(|x| + 1)$. The plot on the right shows the distribution of the transformed time difference.

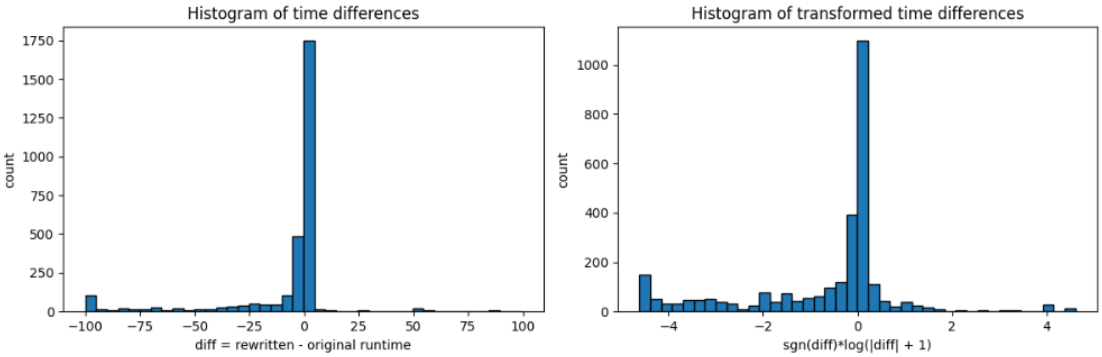


Figure 8.4: Distribution of the regression response = time difference = rewritten runtime - original runtime for PostgreSQL.

Now the ML models can be applied and the results are given and discussed in the following subsections for the train-test split (to be able to compare all models), once with the basic features and once with the additional POS features.

8.1.1 Basic features

All ML models with all hyperparameters for the different classification models and the regression model have been executed. To save some space we decided to provide the results for each type of model, but just with the best hyperparameter combination in Table 8.2. Additionally, we only show the 3-class classification with the 0.5 equal threshold, since the results of the 3-class classification of all other thresholds are always worse than the 2-class classification. The full results are provided in the Appendix (Table A.1 - Table A.3).

For the classifications we use the accuracy and for the regression the MSE as metric. Additionally, we highlight the best model for each type of problem. For the 2-class and the 3-class case the combined neural network of MLP+HGNN achieves the best (highest) accuracy values, where the 3-class classification is the better one in this case. The random forest has the best (smallest) MSE in the regression case. Overall, the decision tree, random forest and the combi model are the best performing models, which is why we will use them as "inspection models" in the following and look at them in more detail.

Model	2-class: Acc	3-class (0.5): Acc	time diff: MSE
5-NN	79.66%	83.45%	0.5850
Decision tree	82.41%	86.21%	0.5809
Rand. forest	82.76%	84.48%	**0.5560**
SVM linear	rbf: 71.03%	linear: 73.10%	rbf: 1.6447
MLP	17-60-40-2: 79.66%	17-60-40-3: 78.97%	17-60-40-1: 1.1265
MLP small	7-60-40-2,max: 78.97%	7-60-40-3,max: 78.97%	7-60-40-1,max: 1.2271
MLP custom	7-60-40-2: 78.97%	7-60-40-3: 80.34%	7-60-40-1: 1.1578
HGNN	1-16-32-2: 81.38%	1-4-16-3: 84.14%	1-32-64-1: 0.8142
Combi	17-60-40-10/1-16-32-10/ 20-40-2: **86.21%**	17-60-40-10/1-16-32-10/ 20-40-3: **90.69%**	17-60-40-10/1-32-64-10/ 20-40-1: 0.5856

Table 8.2: Metrics of ML models for PostgreSQL with basic features (train-test split).

Regression as decision: For the regression we use the MSE as most important metric and predict the runtime as numerical value. Since our task is still to make a decision, we have to define a splitting value, where all values above are predicted as original and below as rewritten. For the regression response we use the time difference such that the original runtime is subtracted from the rewritten runtime and then transformed as explained above.

$$\text{time\_diff} = \text{rewritten\_runtime} - \text{original\_runtime}$$
$$transform(\text{time\_diff}) = sgn(\text{time\_diff}) * log(|\text{time\_diff}| + 1)$$

Similar to the idea of the equal class, we decide to choose the split such that if the predicted original runtime is faster or only a little bit slower then the predicted rewritten runtime, it should be predicted as original. This is the same as if the rewritten runtime plus a treshold is smaller than the original runtime, the prediction should be rewriting. Calling the split γ, this leads to a prediction of "rewr", if the time difference is smaller than $-\gamma$.

$$\begin{aligned} \text{rewritten\_runtime} + \gamma < \text{original\_runtime} \Rightarrow \text{label} = rewr = 1 \Leftrightarrow \\ \text{rewritten\_runtime} - \text{original\_runtime} < -\gamma \Rightarrow \text{label} = rewr = 1 \Leftrightarrow \\ \text{time\_diff} < -\gamma \Rightarrow \text{label} = rewr = 1 \end{aligned}$$

Afterwards the splitting value also has to be transformed to be able to make a decision based on the predicted (transformed) time difference of the regression:

$$\begin{aligned} transform(\text{time\_diff}) < transform(-\gamma) \Rightarrow \text{label} = rewr = 1 \Leftrightarrow \\ transform(\text{time\_diff}) < sgn(-\gamma) * log(|-\gamma| + 1) \Rightarrow \text{label} = rewr = 1 \end{aligned}$$

We decided to try out three different γ values: 0.5, 0.1 and 0.01. In Table 8.3 we can see the accuracy of the performance of the regression model with the different splits. We use the three models decision tree, random forest and combi as explained above and compare the results to the 2-class results, which are provided in italic.

	Acc
Dec. Tree (Class)	*82.41*
Dec. Tree (Reg with split 0.5)	74.48
Dec. Tree (Reg with split 0.1)	78.28
Dec. Tree (Reg with split 0.01)	82.76
Rand. Forest (Class)	*82.76*
Rand. Forest (Reg with split 0.5)	74.14
Rand. Forest (Reg with split 0.1)	76.90
Rand. Forest (Reg with split 0.01)	82.07
Combi (Class)	*86.21*
Combi (Reg with split 0.5)	75.17
Combi (Reg with split 0.1)	79.66
Combi (Reg with split 0.01)	73.45

Table 8.3: Performance of regression with split as classification in comparison to the classification for PostgreSQL with basic features (train-test split).

For the decision tree and random forest the version with $\gamma = 0.01$ is close to or slightly better than the classification version, but overall the regression as basis for the classification is not performing better than the direct 2-class classification independent of the splitting value.

As next step we want to inspect the "inspection models" in more detail. Since we saw that the 2-class classification is working very well and is most suitable for our task, we focus on this case. Therefore, we provide the additional metrics precision and recall and then provide the time differences of the misclassifications of these models.

Avoiding FP: The overall task is to find a decision model, which is able to decide if a query should be executed against a DBMS in its original form or using the rewriting. Therefore, we have two labels: "orig" or 0 if the original runtime is faster and "rewr" or 1 in the other case. We want to improve the existing DBMSs, which means we want to make the query execution faster using the rewriting, if possible. We absolutely do not want to predict that the rewriting version is better, if it is not.

		Predicted	
		rewr/1	orig/0
True	rewr/1	TP	FN
	orig/0	FP	TN

Table 8.4: Confusion matrix.

Given the confusion matrix in Table 8.4 we want to avoid FP as often as possible, because for that case the prediction is "rewr", even if the truth is "orig". This also means the precision is more important than the recall for our application. This is important for the inspection of the misclassifications.

In Table 8.5 we can see the metrics for our inspection models for the 2-class classification.

Model	**Acc**	**Prec**	**Rec**
Decision tree	82.41%	93.08%	74.34%
Rand. forest	82.76%	91.34%	76.69%
Combi	86.21%	90.20%	84.66%

Table 8.5: Accuracy, Precision and Recall for inspection models for PostgreSQL with basic features (train-test split).

We have already seen the accuracy scores before, where the combi model is the best among these three. But now we also provide the other metrics and we can see that the decision tree achieves the best precision, which is an important value for our task as just explained. The recall on the other hand is best for the combi model, which is due to the trade-off relationship between precision and recall (see Section 6.3). But again, precision is more important for our application.

After this quantitative analysis, we do the qualitative analysis by looking at the misclassifications in more detail. As explained the FP are the misclassifications, which we try to avoid. We look at how big the time differences are for the misclassification, since if the label is predicted wrongly, but the time difference is very small anyway, it does not make much difference, which version (original or rewritten) is used. Table 8.6 shows the misclassifications for our inspection models split into orders of magnitude (in seconds) and Figure 8.5 visualizes the distribution of the misclassification with color encoding FP and FN.

Model	**Misclass.**	0.01	0.1	1	10	100	TO
Decision tree	FP	1	6	3	0	0	0
	FN	1	12	11	3	2	0
Rand. forest	FP	0	9	3	0	0	0
	FN	6	12	12	3	5	0
Combi	FP	0	12	3	0	0	0
	FN	3	8	8	2	4	0

Table 8.6: Order of magnitude (in seconds) of the time difference of misclassifications for the inspection models for PostgreSQL with basic features (train-test split).

This result is what we wanted to achieve, since even if there are some FP, their time differences are very small. This means we predict that we should rewrite, even if the original version is faster, which we want to avoid in general, but the rewritten version is only slightly slower, which then is no big deal. Comparing the performance of the different models, the decision tree works best here, since it has the smallest amount of FP and the smallest amount of misclassifications in general. The combi model is the worst one of these three with regard to the misclassifications, but still, it is performing quite well.

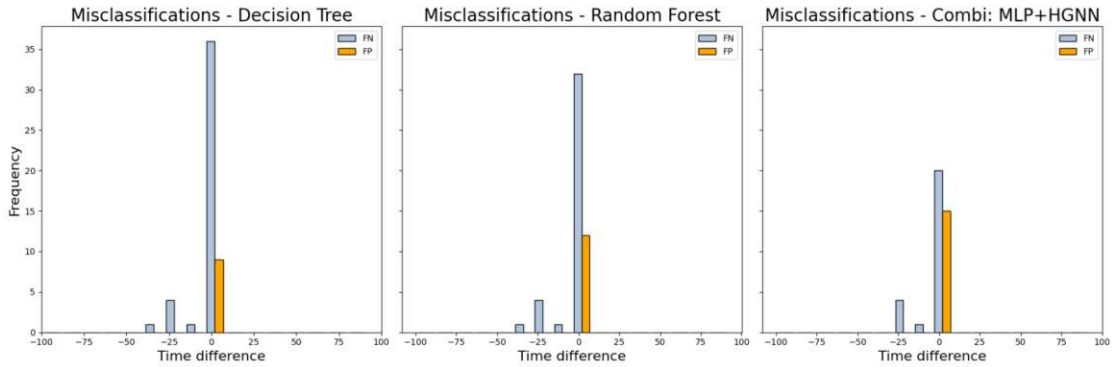


Figure 8.5: Distribution of misclassifications for the inspection models for PostgreSQL with basic features (train-test split).

Since avoiding FP and achieving a high precision, while still having a good accuracy is our goal, the "best" model for PostgreSQL with basic features is the decision tree, which is also nicely interpretable.

8.1.2 Basic features + POS features

Now we want to add some features, namely the POS features (total cost, table rows and join rows), which we get by PostgreSQL EXPLAIN, and hopefully achieve better results using this additional information.

In Table 8.7 the metrics are given for each type of model and the different classifications and regression as above. Again, the full results are provided in the Appendix (Table A.4 - Table A.6).

Model	**2-class: Acc**	**3-class (0.5): Acc**	**time diff: MSE**
5-NN	88.62%	88.62%	0.2581
Decision tree	92.76%	94.48%	0.0846
Rand. forest	**94.48%**	**95.52%**	**0.0661**
SVM linear	rbf: 86.21%	poly: 86.21%	rbf: 0.9154
MLP	30-60-40-2: 91.72%	30-60-40-3: 88.97%	30-80-50-1: 0.4509
MLP small	10-60-40-2,max: 88.97%	10-60-40-3,max: 89.66%	10-80-50-1,med: 0.7646
MLP custom	10-60-40-2: 90.69%	10-60-40-3: 91.38%	10-80-50-1: 0.5671
HGNN	1-16-32-2: 81.38%	1-4-16-3: 84.14%	1-32-64-1: 0.8142
Combi	30-60-40-10/1-16-32-10/ 20-60-20-2: 82.76%	30-60-40-3/1-4-16-3/ 6-3: 78.28%	-

Table 8.7: Metrics of ML models for PostgreSQL with basic features+POS features (train-test split).

We can see that the accuracy for the decision tree and random forest increased a lot, since for the basic version we only got about 82% accuracy and now even 94% for the

2-class classification. In contrast to that the combi model is worse with 82% now and 86% before. We observe a similar behaviour for the 3-class classification and the regression, where the decision tree and random forest outperform the versions before, but the combi model got worse.

Here we have to mention, that the combi model could not be trained for the regression. This is probably due to a vanishing gradient, where multiple methods exist in the literature, but would go beyond the scope of the thesis to adjust the model accordingly, since it only happens twice (for the combi model for regression for PostgreSQL with additional features and SparkSQL with additional features).

Overall, the highest accuracy for this version with the additional features (random forest: 94.48%) is higher than the highest accuracy before with only the basic features (combi: 86.21%).

Again, the next step is to observe how the regression with a splitting value performs as classification in comparison to the 2-class classification. In Table 8.8 we can see the accuracy for the decision tree and random forest (the combi model did not work as just explained).

	Acc
Dec. Tree (Class)	*92.76*
Dec. Tree (0.5)	76.55
Dec. Tree (0.1)	88.28
Dec. Tree (0.01)	92.07
Rand. Forest (Class)	*94.48*
Rand. Forest (0.5)	78.28
Rand. Forest (0.1)	88.97
Rand. Forest (0.01)	92.41

Table 8.8: Performance of regression with split as classification in comparison to the classification for PostgreSQL with basic features+POS features (train-test split).

Both models with each splitting value could not achieve an accuracy as high as the one of the classification, even if it gets close for the 0.01 split.

As inspection models we again use the decision tree, random forest and combi model, due to their good performance and for comparability. In Table 8.9 the accuracy is shown together with the precision and recall scores.

Here the random forest is clearly the best performing model based on the metrics, since it achieves the highest values for all three metrics. The decision tree is only slightly worse and achieves nearly as good results as the random forest. Overall, the accuracy, precision and recall are higher for all these models than for the models with only the basic features before.

Looking at the qualitative results, we can see a similar pattern. Table 8.10 and Figure 8.6 show that the random forest has the fewest FP and fewest misclassifications overall and

Model	Acc	Prec	Rec
Decision tree	92.76%	96.71%	90.18%
Rand. forest	94.48%	97.42%	92.64%
Combi	82.76%	86.45%	82.21%

Table 8.9: Accuracy, Precision and Recall for inspection models for PostgreSQL with basic features+POS features (train-test split).

the decision tree performs nearly as well. The combi model is worse considering the metrics and has much more misclassified values, even a lot of FP. Nearly every time difference for the misclassifications is below 1 second and the majority below 0.1 seconds, which is very nice, because only close decisions are wrongly classified. The number of misclassifications and FP values are smaller for each model with the additional features than for the model with only the basic features.

Model	Misclass.	0.01	0.1	1	10	100	TO
Decision tree	FP	0	4	1	0	0	0
	FN	7	7	2	0	0	0
Rand. forest	FP	0	3	1	0	0	0
	FN	6	5	1	0	0	0
Combi	FP	1	15	3	2	0	0
	FN	5	10	11	3	0	0

Table 8.10: Order of magnitude (in seconds) of the time difference of misclassifications for the inspection models for PostgreSQL with basic features+POS features (train-test split).

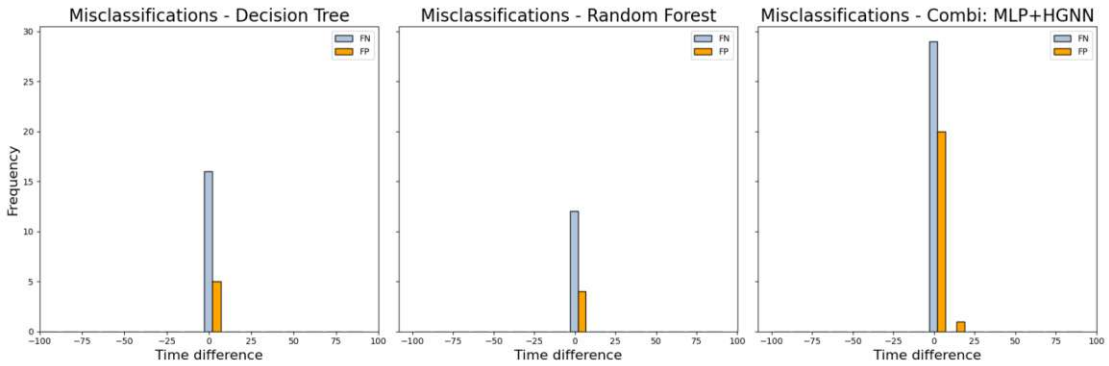


Figure 8.6: Distribution of misclassifications for the inspection models for PostgreSQL with basic features+POS features (train-test split).

Therefore, the best performing model for PostgreSQL with basic features and POS features is the random forest, closely followed by the decision tree. So, it can be discussed if the "best" model is the best performing one, namely the random forest, or only slightly

worse results together with the better interpretability and the less computational effort of the decision tree is the better choice for practice. In this case we would rather opt for the decision tree. Moreover, most of the models with additional features perform better than the ones with only the basic features in terms of metrics and misclassifications.

8.2 DuckDB

This section follows the same procedure as before, but now the evaluation of the queries (original and rewritten version) has been done on the DBMS DuckDB. When looking at the distribution of the runtimes in orders of magnitude (Figure 8.7), we can see that the number of timeouts is much higher for the original queries than for the rewritten. The SNAP queries seem to be harder to evaluate for DuckDB, but the JOB queries seem to be easier in their original form than in their rewritten form. The number of very fast evaluations is not big for either of the two versions.

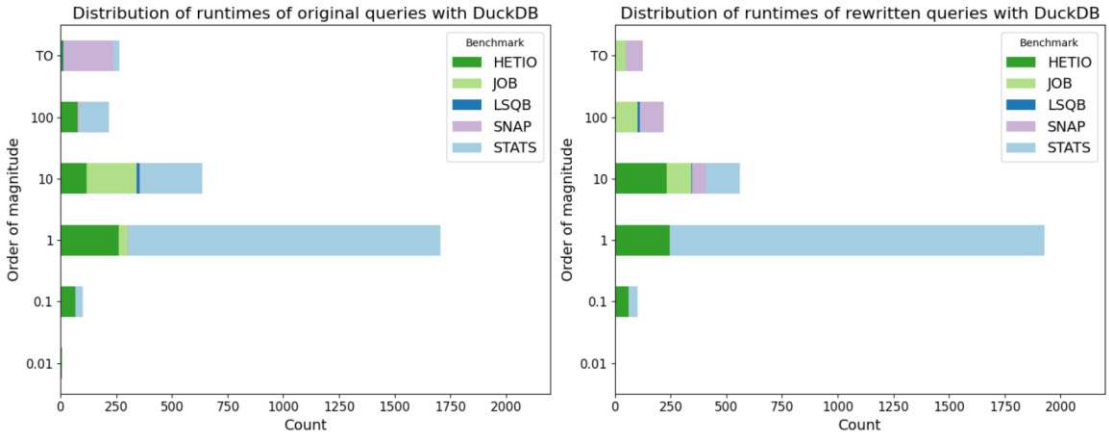


Figure 8.7: Comparison of distributions of orders of magnitude for DuckDB.

In Table 8.11 we can also see that the number of queries, where the evaluation is faster in their rewritten form, is now bigger than the original ones. On the other hand, a lot of these queries, where the rewritten version is faster, are only slightly faster. This can be seen in the table, because a lot of the rewritten queries are shifted to the equal classes (especially for the 0.5 threshold).

	2 classes	3 classes (0.01)	3 classes (0.05)	3 classes (0.1)	3 classes (0.5)
orig	1319	1276	1113	1009	797
rewr	1542	1503	1292	1100	549
equal	-	82	456	752	1515

Table 8.11: Distribution of the classes for the classifications for DuckDB.

The time difference is transformed with the same formula as above ($x_{new} = sgn(x) *$

$log(|x|+1))$ to get the regression response. The distribution of the time difference before and after the transformation can be seen in Figure 8.8.

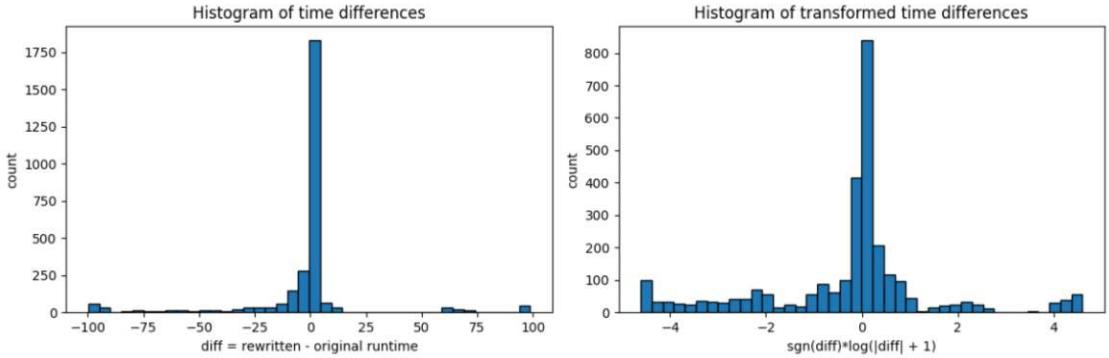


Figure 8.8: Distribution of the regression response = time difference = rewritten runtime - original runtime for DuckDB.

8.2.1 Basic features

As first part we observe the results the models can achieve with the basic features. In Table 8.12 the accuracy or MSE is given for the 2-class and 3-class classification or the regression respectively (for the full results see Appendix Table A.7 - Table A.9).

Model	**2-class: Acc**	**3-class (0.5): Acc**	**time diff: MSE**
5-NN	81.82%	77.27%	0.5873
Decision tree	83.22%	80.77%	0.4905
Rand. forest	83.57%	81.47%	0.4900
SVM linear	rbf: 72.03%	rbf: 74.13%	rbf: 1.6184
MLP	17-80-50-2: 78.32%	17-60-20-3: 79.02%	17-60-40-1: 1.0096
MLP small	7-80-50-2,max: 75.87%	7-80-50-3,max: 79.37%	7-60-40-1,max: 1.253
MLP custom	7-80-50-2: 77.27%	7-80-50-3: 78.32%	7-60-40-1: 1.134
HGNN	1-32-64-2: 84.27%	1-4-16-3: 84.62%	1-32-64-1: 0.5969
Combi	17-80-50-5/1-32-64-5/ 10-20-2: **86.36%**	17-80-50-10/1-16-32-10/ 20-40-3: **88.46%**	17-60-40-10/1-32-64-10/ 20-60-20-1: **0.3778**

Table 8.12: Metrics of ML models for DuckDB with basic features (train-test split).

In this case the combination of the MLP and HGNN performs best for all three cases. The HGNN alone also achieves a similar accuracy for the classification models. Otherwise the decision tree and random forest are again the best performing not deep learning models.

The regression split up into two classes at a value of 0.01 achieves the same (or slightly better) results for the decision tree and random forest, but is not as good for the combi model (Table 8.13).

	Acc
Dec. Tree (Class)	*83.22*
Dec. Tree (Reg with split 0.5)	78.67
Dec. Tree (Reg with split 0.1)	80.42
Dec. Tree (Reg with split 0.01)	83.22
Rand. Forest (Class)	*83.57*
Rand. Forest (Reg with split 0.5)	78.67
Rand. Forest (Reg with split 0.1)	80.77
Rand. Forest (Reg with split 0.01)	83.92
Combi (Class)	*86.36*
Combi (Reg with split 0.5)	80.07
Combi (Reg with split 0.1)	81.82
Combi (Reg with split 0.01)	79.02

Table 8.13: Performance of regression with split as classification in comparison to the classification for DuckDB with basic features (train-test split).

Therefore, we again choose the decision tree, random forest and combi model as inspection models with the 2-class case. In Table 8.14 they are provided together with their precision and recall. In this case the combi model has the highest values for all three metrics. The decision tree is slightly worse, but especially the precision is similar to the combi model.

Model	**Acc**	**Prec**	**Rec**
Decision tree	83.22%	84.17%	77.69%
Rand. forest	83.57%	83.74%	79.23%
Combi	86.36%	86.40%	83.08%

Table 8.14: Accuracy, Precision and Recall for inspection models for DuckDB with basic features (train-test split).

The time difference of the two versions is small for most of the misclassifications for all three models. Table 8.15 and Figure 8.9 provide and visualize the data for us.

Model	**Misclass.**	0.01	0.1	1	10	100	TO
Decision tree	FP	2	7	8	2	0	0
	FN	2	9	11	4	3	0
Rand. forest	FP	2	8	8	2	0	0
	FN	2	7	11	4	3	0
Combi	FP	3	6	5	3	0	0
	FN	2	6	9	4	1	0

Table 8.15: Order of magnitude (in seconds) of the time difference of misclassifications for the inspection models for DuckDB with basic features (train-test split).

The number of FP are always fewer than the amount of FN, which is what we want to have, since FP are the worse misclassifications for our application.

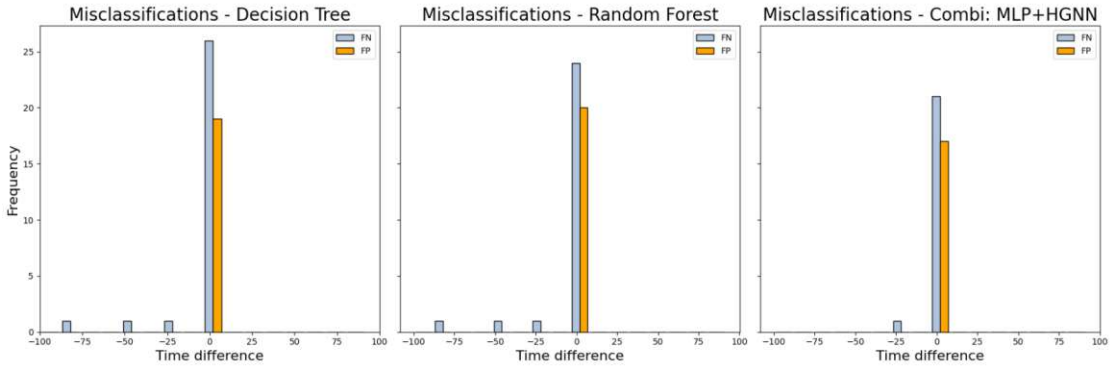


Figure 8.9: Distribution of misclassifications for the inspection models for DuckDB with basic features (train-test split).

For DuckDB with basic features the combi model is performing best based on the metrics and misclassifications. Nevertheless, the combi model is not interpretable, highly complex and needs much more time for the computation as e.g. the decision tree. Additionally, the decision tree performs not much worse, especially regarding the precision and misclassifications (FP).

8.2.2 Basic features + DDB features

To the basic features we now add the cardinality estimate, which we get with the DuckDB EXPLAIN command. This should help the models to improve the predictions due to additional (hopefully useful) information. The important results of the models are provided in Table 8.16 for classification and regression based on their metrics (complete results: Appendix Table A.10 - Table A.12).

Model	**2-class: Acc**	**3-class (0.5): Acc**	**time diff: MSE**
5-NN	86.01%	87.76%	0.3076
Decision tree	87.06%	**90.21%**	0.2086
Rand. forest	**87.76%**	89.86%	**0.2055**
SVM linear	linear: 73.08%	poly: 81.82%	poly: 1.0955
MLP	23-60-20-2: 80.77%	23-60-40-3: 84.97%	23-80-50-1: 0.4525
MLP small	8-60-20-2,max: 76.92%	8-60-40-3,mean: 82.87%	8-80-50-1,min: 0.7122
MLP custom	8-60-20-2: 85.66%	8-60-40-3: 86.36%	8-80-50-1: 0.5016
HGNN	1-16-32-16-2: 87.41%	1-32-64-3: 83.92%	1-32-64-1: 0.4625
Combi	23-60-20-10/1-16-32-16-10/ 20-40-2: 86.36%	23-60-40-5/1-32-64-5/ 10-20-3: 83.92%	23-80-50-10/1-32-64-10/ 20-40-1: 0.4777

Table 8.16: Metrics of ML models for DuckDB with basic features+DDB features (train-test split).

In this case the decision tree and random forest achieve the best metric scores. These two models perform very similarly here and outperform the others. The 3-class classification achieves a higher accuracy for the decision tree and random forest, but is worse for the combi model than the 2-class version. In comparison to the models with only the basic features some models could improve a lot. The decision tree and random forest had an accuracy of about 83% before and increased to about 87% now for the 2-class case. For the 3-class classification these two models achieve accuracy value of about 10% higher with the additional features (80% → 90%) and the MSE is less than the half of the MSE before for the regression model (0.49 → 0.21). The combi model achieves the same result as before for the 2-class case, but gets worse for the 3-class and time difference cases.

The regression with splitting value 0.01 can achieve about the same accuracy as the 2-class classification, but the other splitting values and the combi model are worse, which can be seen in Table 8.17.

	Acc
Dec. Tree (Class)	*87.06*
Dec. Tree (Reg with split 0.5)	79.37
Dec. Tree (Reg with split 0.1)	81.82
Dec. Tree (Reg with split 0.01)	87.76
Rand. Forest (Class)	*87.76*
Rand. Forest (Reg with split 0.5)	79.37
Rand. Forest (Reg with split 0.1)	81.47
Rand. Forest (Reg with split 0.01)	87.06
Combi (Class)	*86.36*
Combi (Reg with split 0.5)	76.92
Combi (Reg with split 0.1)	80.77
Combi (Reg with split 0.01)	79.02

Table 8.17: Performance of regression with split as classification in comparison to the classification for DuckDB with basic features+DDB features (train-test split).

The additional metrics show, that the decision tree achieves the highest precision score, but also a lower recall, wheres the random forest has a slightly better accuracy. The combi model has a bit smaller accuracy score and a smaller precision, but a better recall than the decision tree (Table 8.18).

Model	**Acc**	**Prec**	**Rec**
Decision tree	87.06%	89.92%	81.06%
Rand. forest	87.76%	88.80%	84.09%
Combi	86.36%	86.05%	84.09%

Table 8.18: Accuracy, Precision and Recall for inspection models for DuckDB with basic features+DDB features (train-test split).

The precision and recall values are higher for all three models with the additional features than the metrics for the models with the basic features.

When observing the time differences of the misclassifications in Table 8.19 and Figure 8.10 we can see that there are fewer FP than FN and most FP have small time difference values. Again, this means that the label is wrong, but both versions have a similar runtime anyway. For the decision tree and the random forest the number of misclassifications and also the amount of FP decreased for the models with the additional features.

Model	**Misclass.**	0.01	0.1	1	10	100	TO
Decision tree	FP	1	6	4	1	0	0
	FN	2	15	5	2	1	0
Rand. forest	FP	1	7	5	1	0	0
	FN	2	12	4	2	1	0
Combi	FP	0	4	13	1	0	0
	FN	1	9	8	3	0	0

Table 8.19: Order of magnitude of the time difference of misclassifications for the inspection models for DuckDB with basic features+DDB features (train-test split).

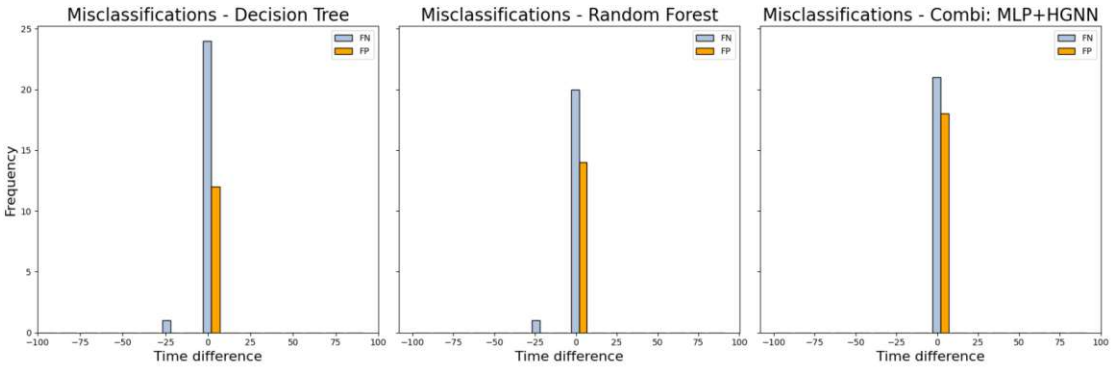


Figure 8.10: Distribution of misclassifications for the inspection models for DuckDB with basic features+DDB features (train-test split).

We can see that the decision tree is probably the best model here together with the random forest. Even if the accuracy for the random forest is sligthly better than for the decision tree, the precision is higher for the decision tree and it is the easier and more interpretable model. Overall, the models with the additional features outperform the models with the basic features, especially for the decision tree and random forest.

8.3 SparkSQL

The third DBMS we are looking at, is SparkSQL, which has been used to run the queries and provide the runtimes for the ML models. In the beginning we observe the distribution

of the runtimes for the original and rewritten queries provided in orders of magnitude in Figure 8.11.

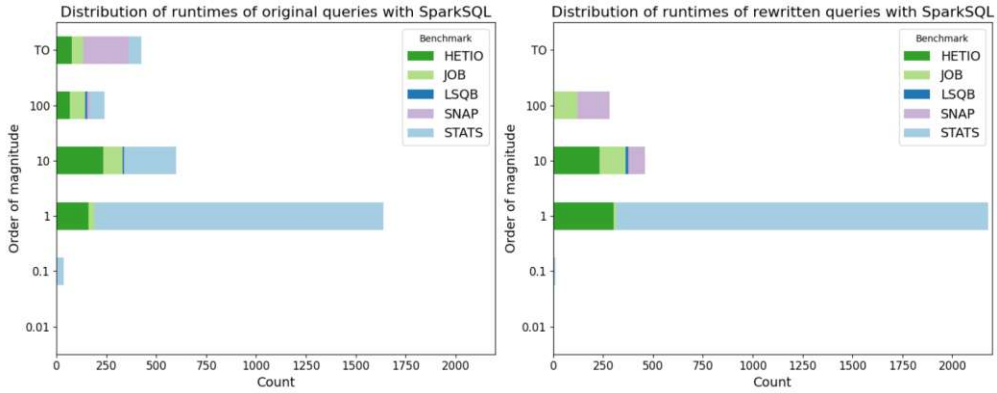


Figure 8.11: Comparison of distributions of orders of magnitude for SparkSQL.

The number of timeouts is much bigger for the queries evaluated with the original form, whereas the number of queries evaluated fast is a bit higher for the original queries. However, the amount of queries evaluated very fast is not big for either of the two versions. For all benchmark datasets there are parts of them, where the evaluation gets faster using the rewriting method.

In Figure 8.12 we can see the distribution of the regression response. On the left we can see the skewed distribution with a wide range, where most of the values are in the middle. As before we transform the time difference with this formula $x_{new} = sgn(x) * log(|x| + 1)$ to achieve a distribution closer to a normal distribution.

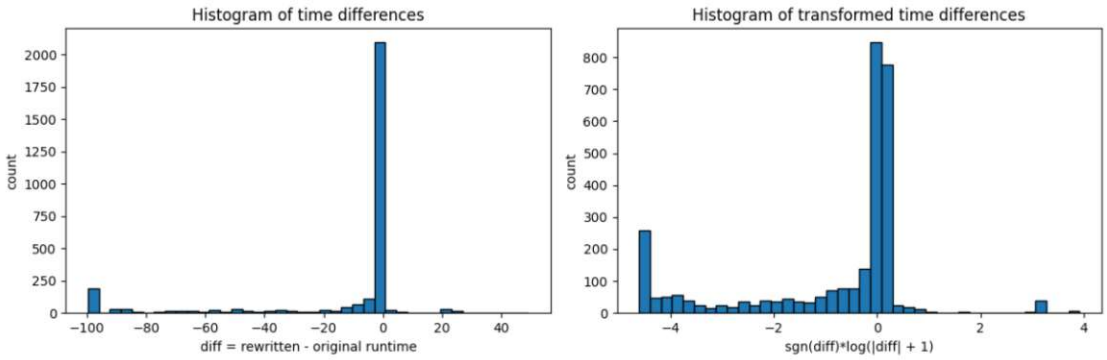


Figure 8.12: Distribution of the regression response = time difference = rewritten runtime - original runtime for SparkSQL.

In Table 8.20 we can see that about half of the queries have a faster evaluation in the original form and the other half in the rewritten form. If the original query is faster, most of the times it is only a little bit faster, since they are mostly in the equal class with threshold 0.5 then.

	2 classes	3 classes (0.01)	3 classes (0.05)	3 classes (0.1)	3 classes (0.5)
orig	1452	1412	1146	726	104
rewr	1482	1444	1351	1260	1028
equal	-	78	437	948	1802

Table 8.20: Distribution of the classes for the classifications for SparkSQL.

8.3.1 Basic features

For the models using the basic features we can see the results in Table 8.21 (for full results see Appendix Table A.13 - Table A.15).

Model	**2-class: Acc**	**3-class (0.5): Acc**	**time diff: MSE**
5-NN	82.94%	86.35%	0.6731
Decision tree	83.62%	89.08%	0.5790
Rand. forest	83.28%	89.08%	0.5846
SVM linear	rbf: 71.33%	rbf: 79.86%	rbf: 1.6123
MLP	17-80-50-2: 77.47%	17-60-40-3: 84.98%	17-80-50-1: 1.1241
MLP small	7-80-50-2,med: 76.79%	7-80-50-3,max: 85.67%	7-80-50-1,max: 1.2027
MLP custom	7-80-50-2: 76.79%	7-80-50-3: 86.01%	7-80-50-1: 1.1544
HGNN	1-16-32-2: 80.55%	1-32-64-3: 88.40%	1-32-64-1: 0.6577
Combi	17-80-50-10/1-16-32-10/ 20-60-20-2: **83.96%**	17-80-50-5/1-4-16-5 10-20-3: **91.13%**	17-80-50-10/1-32-64-10/ 20-40-1: **0.4346**

Table 8.21: Metrics of ML models for SparkSQL with basic features (train-test split).

For the 2-class classification the decision tree, random forest and combi model perform very similarly. The 3-class version achieves a higher accuracy for all models than the 2-class case, where the combi model is the best, closely followed by the decision tree and random forest. For the regression the combi model has the smallest MSE. The next best models are the decision tree and random forest.

When inspecting the three models decision tree, random forest and the combi model in more detail, we can see that the decision tree achieves the highest precision value (which is important for us). The other two models are not much worse and achieve a better recall. This can be seen in Table 8.22.

Model	**Acc**	**Prec**	**Rec**
Decision tree	83.62%	86.76%	79.73%
Rand. forest	83.28%	83.67%	83.11%
Combi	83.96%	84.83%	83.11%

Table 8.22: Accuracy, Precision and Recall for inspection models for SparkSQL with basic features (train-test split).

The regression with a splitting value performs as classification about the same as the 2-class classification for the decision tree and random forest with a split of 0.01 and better

results for the 0.1 split. The other split and the combi models are worse than the direct classification (Table 8.23).

	Acc
Dec. Tree (Class)	*83.62*
Dec. Tree (0.5)	79.52
Dec. Tree (0.1)	84.98
Dec. Tree (0.01)	83.62
Rand. Forest (Class)	*83.28*
Rand. Forest (0.5)	79.52
Rand. Forest (0.1)	84.64
Rand. Forest (0.01)	83.62
Combi (Class)	*83.96*
Combi (0.5)	78.84
Combi (0.1)	81.91
Combi (0.01)	75.77

Table 8.23: Performance of regression with split as classification in comparison to the classification for SparkSQL with basic features (train-test split).

We can see the distribution and the splits into orders of magnitude of the time difference of the misclassifications in Table 8.24 and Figure 8.13.

Model	**Misclass.**	0.01	0.1	1	10	100	TO
Decision tree	FP	1	10	6	1	0	0
	FN	6	16	3	3	2	0
Rand. forest	FP	1	14	8	1	0	0
	FN	5	14	1	3	2	0
Combi	FP	2	12	7	1	0	0
	FN	4	14	4	3	0	0

Table 8.24: Order of magnitude of the time difference of misclassifications for the inspection models for SparkSQL with basic features (train-test split).

More than half of the FP for each of the three models have a time difference smaller or equal to 0.01, which means the runtimes are very similar in the two versions (original and rewritten). Most of the FN have a very small time difference, too. The decision tree has the fewest FP values, but slightly more FN.

For SparkSQL with basic features the decision tree achieves the best precision and a similar accuracy as the others. Moreover, the distribution of the time difference for the misclassifications is the best for the decision tree and it is an easy and interpretable model. We consider it the "best" model for this setting.

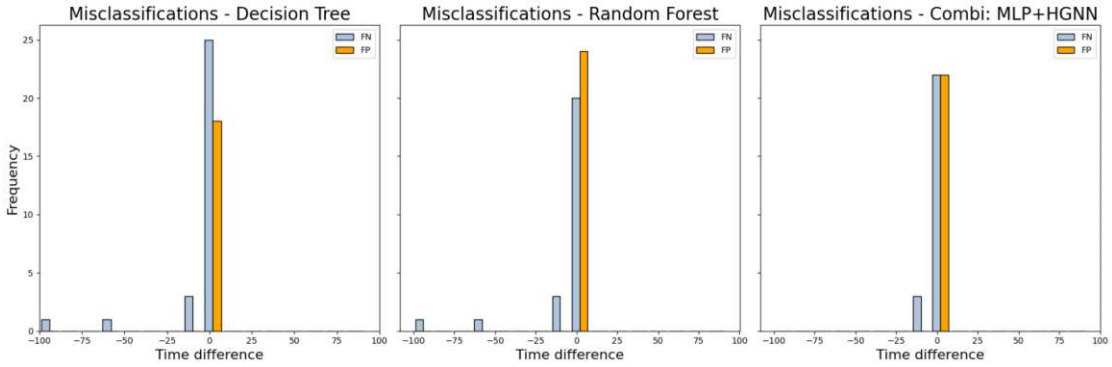


Figure 8.13: Distribution of misclassifications for the inspection models for SparkSQL with basic features (train-test split).

8.3.2 Basic features + POS features

Again, we want to improve the model by adding additional features. Since SparkSQL does not provide additional information, we have decided to use the POS features here. The model results can be seen in Table 8.25 (full results Appendix Table A.13 - Table A.15).

Model	**2-class: Acc**	**3-class (0.5): Acc**	**time diff: MSE**
5-NN	86.35%	94.20%	0.2863
Decision tree	88.74%	96.93%	**0.0222**
Rand. forest	**89.08%**	**97.27%**	0.0252
SVM linear	rbf: 79.52%	poly: 88.74%	rbf: 1.0176
MLP	30-40-10-2: 86.35%	30-80-50-3: 94.54%	30-80-50-1: 0.3607
MLP small	10-40-10-2,max: 83.28%	10-80-50-3,med: 94.54%	10-80-50-1,min: 0.5706
MLP custom	10-40-10-2: 85.32%	10-80-50-3: 92.83%	10-80-50-1: 0.6684
HGNN	1-16-32-2: 80.55%	1-32-64-3: 88.40%	1-32-64-1: 0.6577
Combi	30-40-10-10/1-16-32-10/ 20-40-2: 81.57%	30-80-50-10/1-4-16-10/ 20-40-3: 73.72%	-

Table 8.25: Metrics of ML models for SparkSQL with basic features+POS features (train-test split).

Here the random forest is the best for the classifications, slightly followed by the decision tree and vice versa for the regression. The combi model is much worse here. For the 3-class classification the random forest achieves an accuracy of over 97%. In comparison to the models with the basic features before, these models with the additional features achieve a higher accuracy for the decision tree and random forest (83% → 89%), but a lower for the combi model (84% → 81%).
As explained above we could not run the combi model for this setting for the regression case.

The regression with a splitting value as classification cannot achieve the same accuracy

values as the 2-class classification, as we can see in Table 8.26. Still, the scores are not much worse.

	Acc
Dec. Tree (Class)	*88.74*
Dec. Tree (0.5)	84.64
Dec. Tree (0.1)	88.05
Dec. Tree (0.01)	88.05
Rand. Forest (Class)	*89.08*
Rand. Forest (0.5)	84.30
Rand. Forest (0.1)	88.40
Rand. Forest (0.01)	88.05

Table 8.26: Performance of regression with split as classification in comparison to the classification for SparkSQL with basic features+POS features (train-test split).

Using our inspection models, we can see that the decision tree and random forest achieve precision values of 94% (Table 8.27). This is highly appreciated for our purpose. Moreover, the values of precision and recall are higher for these models with the additional features than the ones with the basic features.

Model	**Acc**	**Prec**	**Rec**
Decision tree	88.74%	94.57%	82.43%
Rand. forest	89.08%	94.62%	83.11%
Combi	81.57%	84.06%	78.38%

Table 8.27: Accuracy, Precision and Recall for inspection models for SparkSQL with basic features+POS features (train-test split).

Only very few FP exist for the decision tree and random forest and all of them have time difference values of under one second. Additionally, most of them and most of the FN even have time differences of under 0.1 seconds, which means the runtimes of the original and rewritten version are almost the same. We can see that in Table 8.28. This misclassification results are again slightly better than for the models without the additional features.

The distributions of the misclassifications for the decision tree and random forest are very close around zero, which we can see in Figure 8.14.

For the SparkSQL runtimes and basic features and POS features the decision tree and random forest achieve very similar results. Since the decision tree has a slightly better precision and is the easier model, we would consider it as the best model here. The models with the additional features outperform the version with only the basic features in terms of metrics and inspection of misclassifications.

Model	Misclass.	0.01	0.1	1	10	100	TO
Decision tree	FP	0	5	2	0	0	0
	FN	5	17	4	0	0	0
Rand. forest	FP	0	4	3	0	0	0
	FN	5	17	3	0	0	0
Combi	FP	2	11	8	1	0	0
	FN	3	13	8	3	5	0

Table 8.28: Order of magnitude of the time difference of misclassifications for the inspection models for SparkSQL with basic features+POS features (train-test split).

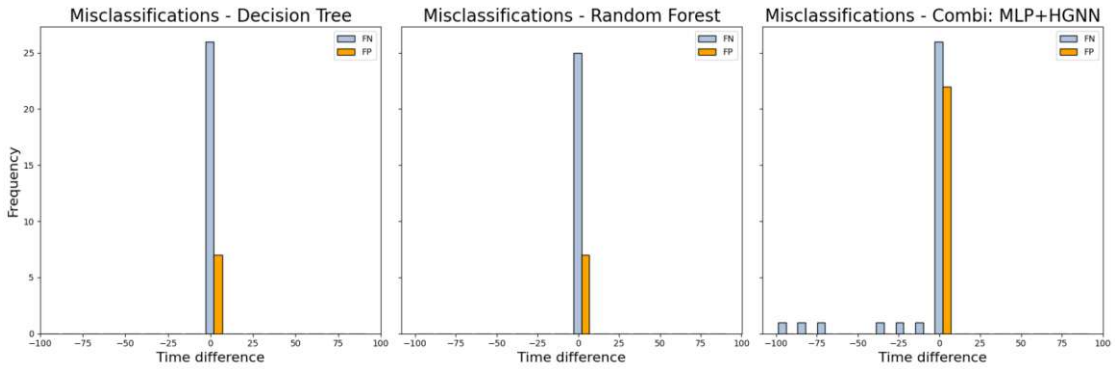


Figure 8.14: Distribution of misclassifications for the inspection models for SparkSQL with basic features+POS features (train-test split).

8.4 Comparison

After observing the results with quantitative and qualitative analysis separately for the data of each DBMS, we now want to compare their results and find out, which model performs best overall and obtains other insights. In the beginning we look at the runtime distributions in orders of magnitude for the original and rewritten evaluation in Table 8.29 once more.

	POS		DDB		SPA	
Runtime	orig eval.	rewr. eval	orig eval.	rewr. eval	orig eval.	rewr. eval
[0,0.01]	115	1	7	0	0	0
(0.01,0.1]	797	522	101	103	35	12
(0.1,1]	1053	1937	1706	1930	1638	2179
(1,10]	434	227	638	560	598	461
(10,100]	244	208	219	219	241	282
TO (>100)	293	41	265	124	424	2

Table 8.29: Runtime distribution in order of magnitudes for both methods and all three DBMSs.

We have already seen these results as plots (Figure 8.3, Figure 8.7, Figure 8.11) for each DBMS separately. Now we want to build a connection between them. We can see that PostgreSQL is the only one with multiple queries evaluated faster than 0.01 seconds, where most of these queries were in their original form. SparkSQL is the DBMS with the slowest runtimes overall and also the one with the most timeouts. The general pattern, that the evaluation of the original queries leads to a broader distribution with some very fast and some very slow evaluations, can be seen for all three DBMSs. In contrast to that the rewritten form leads to fewer timeouts, but also to less fast runtimes. This once more confirms our initial problem: Neither the original nor the rewritten version is the better choice for all queries, but for some cases the original is better and for some cases the rewritten is better. This is the reason why we want to construct a decision program, which tells us which version should be used.

This decision program will be an ML model and we now compare the performances of different models based on the accuracy and MSE. We provide the best three models for each DBMS and feature combination for the 2-class classifications in Table 8.30.

Data	Best	Second best	Third best
POS: basic	Combi (86.21%)	Rand. Forest (82.76%)	Dec. Tree (82.41%)
POS: basic+POS	Rand. Forest (94.48%)	Dec. Tree (92.76%)	MLP (91.72%)
DDB: basic	Combi (86.36%)	HGNN (84.27%)	Rand. Forest (83.57%)
DDB: basic+DDB	Rand. Forest (87.76%)	HGNN (87.41%)	Dec. Tree (87.06%)
SPA: basic	Combi (83.96%)	Dec. Tree (83.62%)	Rand. Forest (83.28%)
SPA: basic+POS	Rand. Forest (89.08%)	Dec. Tree (88.74%)	k-NN (86.35%)

Table 8.30: Best models for 2 classes.

For all DBMSs the combi model achieves the highest accuracy, if the basic features are used. When adding additional features the random forest is the best performing model. Nevertheless, the accuracy is often very similar for each of the best three models and as mentioned in the last sections, the accuracy is not the only important measure. We also have to consider the precision (and recall), as well as the performance of the models, i.e. observing the behaviour of the misclassifications. The decision tree is under the top three models for five out of the six cases of DBMS and features based on the accuracy. Often it achieves one of the best precision scores and the TP misclassifications are often only for cases where the runtimes are similar anyway. Another very important point is that this model is interpretable and the decisions can be explained. Additionally, it is computationally one of the cheaper models. Therefore, we decide that the decision tree is the "best" model for our task, since it achieves very good results with the metrics and misclassifications and is an interpretable model, which can be computed easily.

A similar behaviour can be seen for the 3-class classification with cut-off 0.5 in Table 8.31. The decision tree is under the top 2 in five settings. For the other cut-offs the results are given in A.8, since they do not achieve values as good as with the 0.5 cut-off. The 0.5 cut-off 3-class version achieves higher accuracy values than the 2-class version, but it is not suitable for our task, since we have to decide, which version should be used (original

and rewritten). One could argue, that for the equal class it does not really matter, which version is used, since they have nearly equal runtimes anyway. Then, this 3-class case performs really well and should be used.

Data	Best	Second best	Third best
POS: basic	Combi (90.69%)	Dec. Tree (86.21%)	Rand. Forest (84.48%)
POS: basic+POS	Rand. Forest (95.52%)	Dec. Tree (94.48%)	MLP (91.38%)
DDB: basic	Combi (88.46%)	HGNN (85.66%)	Rand. Forest (81.47%)
DDB: basic+DDB	Dec. Tree (90.21%)	Rand. Forest (89.86%)	k-NN (87.76%)
SPA: basic	Combi (91.13%)	Dec. Tree (89.08%)	Rand. Forest (89.08%)
SPA: basic+POS	Rand. Forest (97.27%)	Dec. Tree (96.93%)	MLP (94.54%)

Table 8.31: Best models for 3 classes with cut-off 0.5.

Finally, we show the best models for the regression case based on the MSE in Table 8.32. Again, the decision tree and random forest are under the top two (or three) models, where the decision tree is the easier model and should be taken. Nevertheless, as we have seen in the last sections, the regression with a splitting value as classification most of the times does not achieve the performance of the 2-class classification and only very few times has a similar or slightly better result.

Data	Best	Second best	Third best
POS: basic	Rand. Forest (0.5560)	Dec. Tree (0.5809)	k-NN (0.5850)
POS: basic+POS	Rand. Forest (0.0661)	Dec. Tree (0.0846)	k-NN (0.2581)
DDB: basic	Combi (0.3778)	Rand. Forest (0.4900)	Dec. Tree (0.4905)
DDB: basic+DDB	Rand. Forest (0.2055)	Dec. Tree (0.2086)	k-NN (0.3076)
SPA: basic	Combi (0.4346)	Dec. Tree (0.5790)	Rand. Forest (0.5846)
SPA: basic+POS	Dec. Tree (0.0222)	Rand. Forest (0.0252)	k-NN (0.2863)

Table 8.32: Best models for time difference.

Additional insights that we get from these comparisons and the last sections are that the models with additional features always perform better than the ones with only the basic features. The setting with the PostgreSQL runtimes with the basic features and POS features achieves the best results of all our models and settings.

8.5 Final model

As final model, which we apply on the untouched test set to see how well the model generalizes, we use the decision tree as argued above. This means we use the decision tree, which we trained on the training set and validated on the validation set (where we saw which model with which parameter setting performed best) and now predict the labels using this decision tree for the test set. This is done for each setting of DBMS runtimes and feature combination.

We now observe the performance based on the metrics and inspect the misclassifications in a similar way as before. In Table 8.33 we can see that the models perform very well on

the test set. All metrics achieve values above 80% and therefore, the model generalizes well. The PostgreSQL runtimes together with the basic features and POS features even achieve an accuracy of about 94%, a precision of 96% and a recall of 91%, which is a very good result. Again, the models with additional features always perform better.

Setting	**Acc**	**Prec**	**Rec**
POS, basic	86.55%	92.13%	80.14%
POS, basis+POS	93.79%	96.38%	91.10%
DDB, basic	85.37%	87.60%	81.29%
DDB, basis+DDB	90.21%	92.00%	86.47%
SPA, basic	80.27%	80.54%	80.54%
SPA, basic+POS	90.14%	91.10%	89.26%

Table 8.33: Accuracy, Precision and Recall for the final model on the test set (train-test split).

Additional to the numbers observed with the metrics, we visualize the distributions of orders of magnitude for each setting for the final test set.

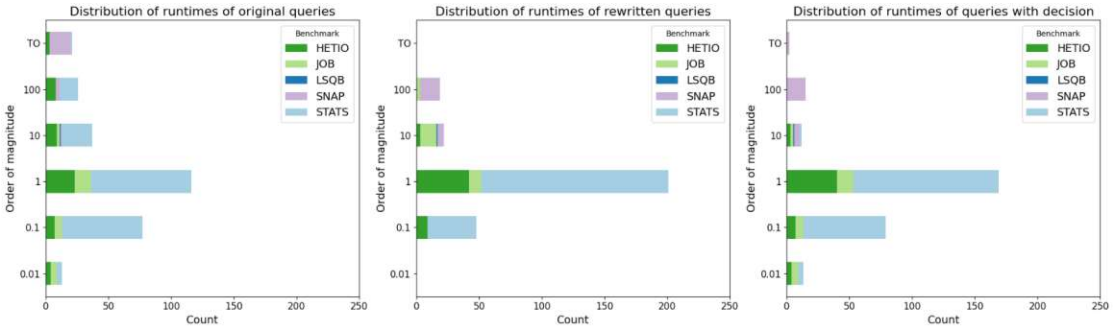


Figure 8.15: Comparison of distributions of orders of magnitude for PostgreSQL with basic features.

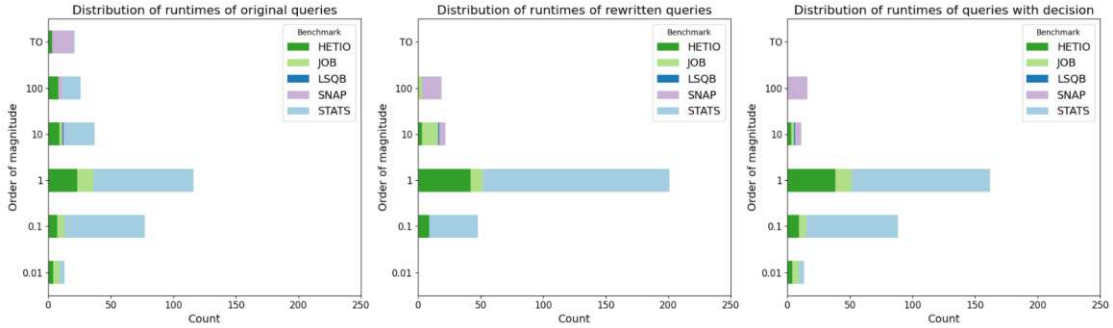


Figure 8.16: Comparison of distributions of orders of magnitude for PostgreSQL with basic features+POS features.

On the left the distribution is provided for the evaluation times of the original queries, in the middle for the rewritten queries and on the right for the distribution of either the original or the rewritten runtime based on the decision we made using our decision tree.

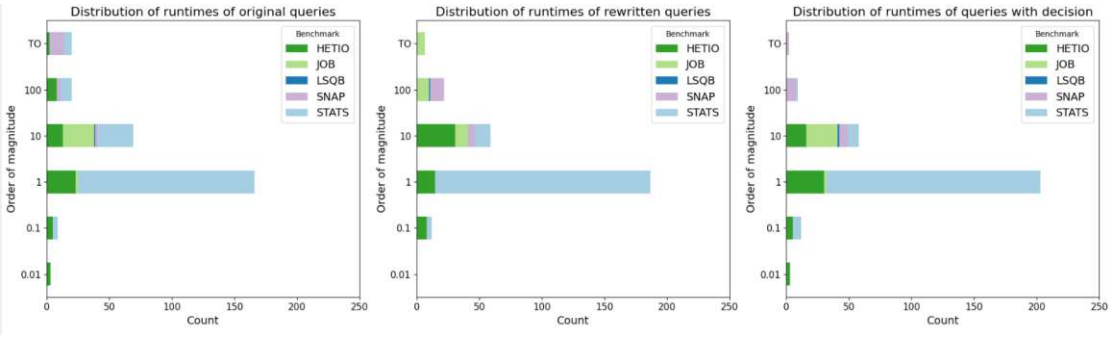


Figure 8.17: Comparison of distributions of orders of magnitude for DuckDB with basic features.

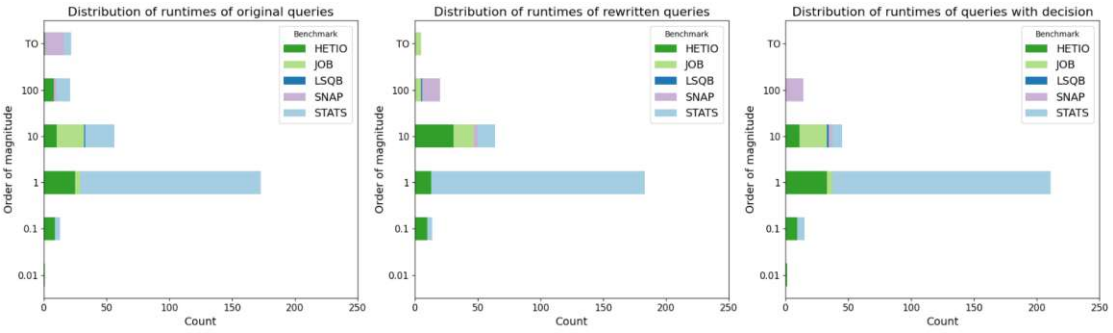


Figure 8.18: Comparison of distributions of orders of magnitude for DuckDB with basic features+DDB features.

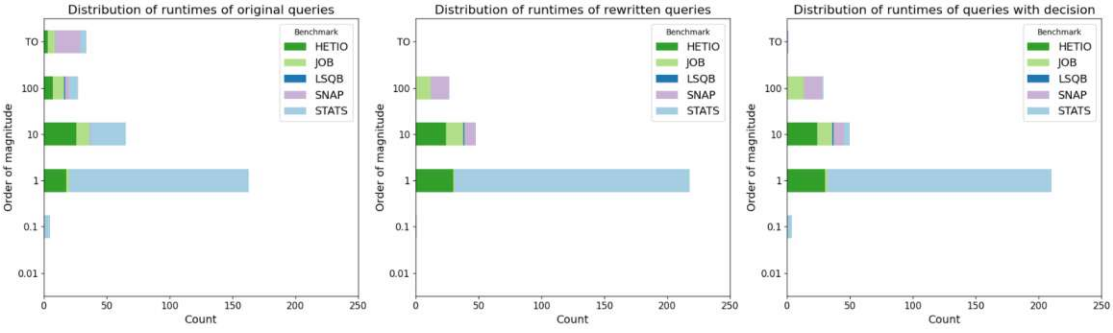


Figure 8.19: Comparison of distributions of orders of magnitude for SparkSQL with basic features.

In Figure 8.15 this is given for PostgreSQL with basic features, in Figure 8.16 for PostgreSQL with additional POS features, in Figure 8.17 for DuckDB with basic features, in Figure 8.18 for DuckDB with additional DDB features, in Figure 8.19 for SparkSQL with basic features and in Figure 8.20 for SparkSQL with additional POS features.

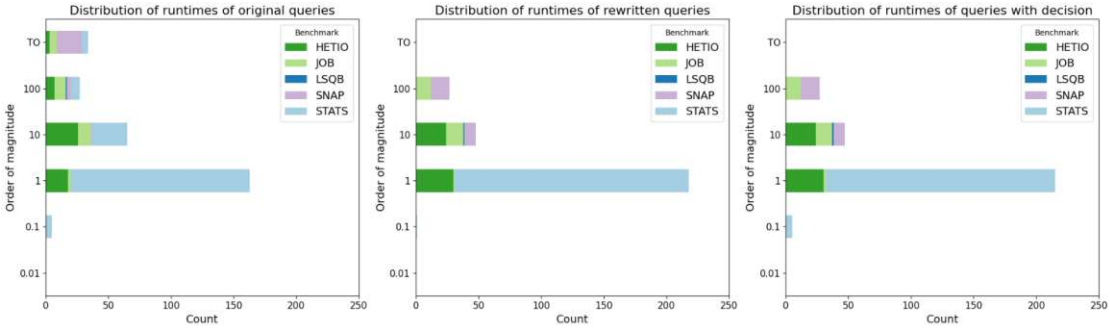


Figure 8.20: Comparison of distributions of orders of magnitude for SparkSQL with basic features+POS features.

For each setting we observe that the number of timeouts and longer evaluation times is reduced and the number of fast evaluation times is increased, which was our goal.

Now, we want to examine the misclassifications (given in Table 8.34). This is a very nice result, too, since for all FP the time difference between the two versions (original and rewritten) is always below 1 second and for most cases even below 0.1 seconds.

Setting	**Misclass.**	0.01	0.1	1	10	100	TO
POS, basic	FP	1	6	3	0	0	0
	FN	1	12	11	3	2	0
POS, basic+POS	FP	1	2	2	0	0	0
	FN	2	6	5	0	0	0
DDB, basic	FP	1	7	5	2	1	0
	FN	2	12	4	5	3	0
DDB, basic+DDB	FP	3	5	2	0	0	0
	FN	2	8	5	3	0	0
SPA, basic	FP	2	15	12	0	0	0
	FN	1	12	8	6	2	0
SPA, basic+POS	FP	1	9	3	0	0	0
	FN	2	10	4	0	0	0

Table 8.34: Order of magnitude of the time difference of misclassifications for the final model on the test set (train-test split).

There are some FN with a bit higher time differences, but the majority is below 1 second. Again, the PostgreSQL model with basic features and POS features performs best (has the fewest misclassifications and fewest FP and all of them have a time difference below 1 second).

Additionally, we provide two statistical tests, which to test the performance of the decision program compared to the plain evaluation. In this case, this means that we test if the mean or median of the runtimes, which we get with our decision program, is significantly smaller than the mean/median of the original runtimes. As the test for the median we use the Wilcoxon sign-rank test and as the test for the mean a paired sample t-test (see Section 7.4.2). If the p-value of the statistical test is below an alpha value, which we chose as 0.1 (a common choice), then the null hypothesis can be rejected. The null hypotheses are that the means or medians of the two groups are the same. So, if we reject that, we can say that the means/medians are significantly different for the two groups and in our case that means, that using the decision program gives a significantly better result than just taking the original queries.

In Table 8.35 we can see that all p-values are very small and we can reject the null every time. This means the means and medians of the two groups are significantly different. Since the means and medians of the runtimes, which we got with our decision program, are smaller and significantly different from those of the original runtimes, we get that the decision program leads to significantly lower means/medians of runtimes on the test set. This is exactly what we wanted to achieve.

Setting	**Test**	**Statistic**	**p-value**	**Reject/not reject**
POS, basic	Wilcoxon test	187.0	$1.0710 * 10^{-20}$	Reject
	Paired t-test	6.2546	$1.4305 * 10^{-9}$	Reject
POS, basic+POS	Wilcoxon test	114.0	$2.5330 * 10^{-23}$	Reject
	Paired t-test	6.4625	$1.0710 * 10^{-10}$	Reject
DDB, basic	Wilcoxon test	639.0	$6.6518 * 10^{-17}$	Reject
	Paired t-test	5.3127	$2.1757 * 10^{-7}$	Reject
DDB, basic+DDB	Wilcoxon test	204.0	$3.6101 * 10^{-20}$	Reject
	Paired t-test	5.5709	$5.8647 * 10^{-8}$	Reject
SPA, basic	Wilcoxon test	696.0	$1.8638 * 10^{-20}$	Reject
	Paired t-test	7.0013	$1.7362 * 10^{-11}$	Reject
SPA, basic+POS	Wilcoxon test	171.0	$3.3822 * 10^{-24}$	Reject
	Paired t-test	7.2394	$3.9816 * 10^{-12}$	Reject

Table 8.35: Statistical tests for the final model on the test set (train-test split).

Finally, we want to visualize the decision trees, which are our final models and try to find out which features were the most important ones. In Figure 8.21 we provide the decision tree of our best performing model (PostgreSQL runtimes with basic features and POS features). The visualizations of the other decision trees can be seen in the Appendix: Figure A.1 to Figure A.5. These visualizations give us an overview of how the structure of the decision trees look like. Nevertheless, since the depth of the six trees are between 15 and 19, it is hard/impossible to read all single decisions.

Therefore, we state the most important features of each tree. Features, which are more often used for decision, and features, which distinguish between the groups well, are

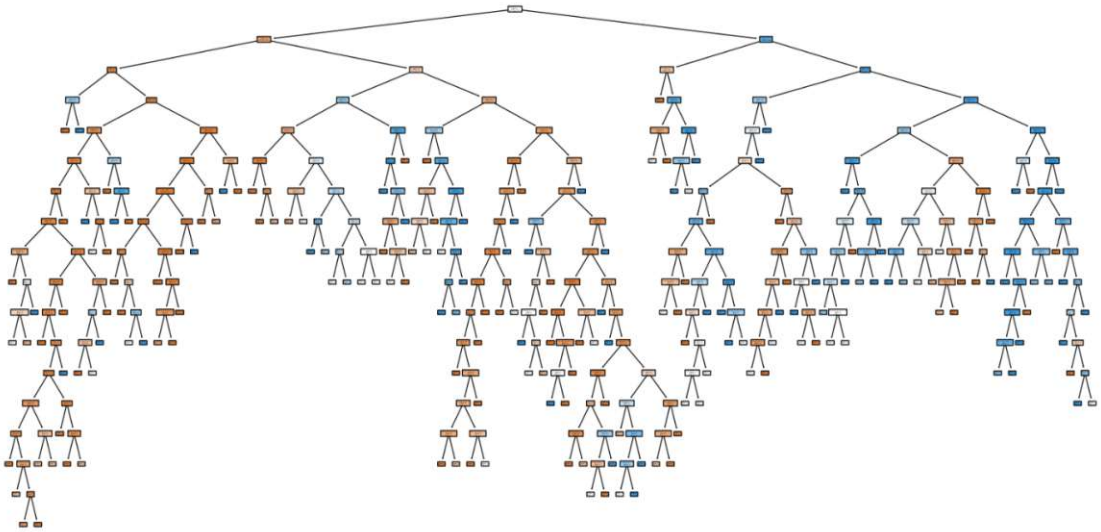


Figure 8.21: Visualization of the final model (=decision tree) for PostgreSQL with basic features+POS features (train-test split).

considered to be more important. This can be measured with the Gini importance, which calculates the Gini impurity for each decision and assigns importances between 0 and 1 to each feature.

In Table 8.36 the three most important features in each tree are given together with the Gini importance value. All features with their importances are given in the Appendix Table A.28.

Setting	**First**	**Second**	**Third**
POS, basic	mean(container c.) 0.488273	# filters 0.235353	# joins 0.091582
POS, basic+POS	max(join rows) 0.494756	min(join rows) 0.116054	q75(join rows) 0.058002
DDB, basic	mean(container c.) 0.408201	# filters 0.211150	# joins 0.095617
DDB, basic+DDB	med(est. cardinality) 0.241326	mean(container c.) 0.112311	max(container c.) 0.110673
SPA, basic	# filters 0.393823	mean(container c.) 0.270023	# conditions 0.120211
SPA, basic+POS	max(join rows) 0.287186	# filters 0.157783	total cost 0.089817

Table 8.36: Three most important features (Gini importance) of final model on the test set for PostgreSQL with basic features (train-test split).

We can see that the feature "# filters", the number of filters in the query, is under the top three most important features for five of our six settings. Additionally, it is interesting to see, that for the three models with additional features, at least one of these additional features is under the top three considering the importance. Since we saw that the models are performing better with additional features, it is not such a big surprise, that they influence the model a lot. For the PostgreSQL setting with POS features all three most important features are based on the POS join row feature. For the DuckDB setting with DDB features, the DDB feature cardinality is the most important feature. For the SparkSQL setting with the additional POS features the join rows and total cost, which are POS features, are the first and third most important feature.

CHAPTER 9

Conclusion

In this thesis we design a decision program, which can be used to decide between two evaluation methods for SQL queries. The one method is the plain evaluation of a query on a DBMS. The other version is a rewriting method introduced in Gottlob et al., 2023. The problem is that neither of these methods is better for all queries, but there are queries, where the original version is faster and queries, where the rewritten version can decrease the evaluation runtime. So, we want to create a decision program based on a Machine Learning model. The models we compare are k-nearest neighbors, decision trees, random forests, support vector machines, multi-layer perceptrons, hypergraph neural networks and a combination of the latter two.

We find out that the decision tree is the "best" model. Therefore, we use it as our decision program. There are multiple factors, which lead to our choice. On the one hand, the decision tree achieves high scores for the metrics (up to over 90% for accuracy, precision and recall). On the other hand, there are only a few misclassifications and most of them are in cases, where the original and rewritten runtimes are very similar anyway. Additionally, the decision tree is a Machine Learning model, which is interpretable and has an easy computation. Furthermore, we can identify features, which are benefical for the model results and we provide the most influential features for each model setting.

For all three DBMSs, namely PostgreSQL, DuckDB and SparkSQL, we can outperform the plain evaluation of the queries, using our decision program and the rewriting method for cases, where it improves the runtime. We apply the final decision program, which means the decision tree, on the unseen test set for our final analysis, to see that it generalizes very well. Additionally, we use two statistical tests to compare the mean/median of the runtimes for the original version and our proposed version. For all DBMSs and all our settings the mean and median of the runtimes of our version are statistical significantly smaller than the mean and median of the original runtimes. We also can identify which features are important for making the decisions.

In future work the whole process could be extended to a broader class of queries, since our approach only works for acyclic queries (0MA queries). For that, the decompositions described in Chapter 5 could be used to split up cyclic queries into acyclic ones, which then could be used in the same way as we did. For cyclic queries this means that the edge cover should be constructed and if all tables in each node of the edge cover are joined together, the resulting tables are an acyclic version of this query. Using the decompositions we would get a join tree of this edge cover and would be able to apply all other steps like for the acyclic queries. The Machine Learning part could also be further extended. One way would be to add and/or construct additional features to the ones we used. Another approach would be to introduce more complex neural networks and use additional message passing layers for the hypergraph neural networks. One interesting thing would be to reduce the features and only use the ones which we got as the most important features in the end and find out how well models based on them would perform.

APPENDIX A

Results of Machine Learning Models

We provide the metrics for all ML models for all data from different DBMSs for all hyerparameter combinations we used for train-test split and cross-validation.
As data we use: PostgreSQL data with basic features, PostgreSQL data with basic features and POS features, DuckDB data with basic features, DuckDB data with basic features and DDB features, SparkSQL data with basic features, SparkSQL data with basic features and POS features.
For each of them we have the models: k-NN, decision tree, random forest, SVM, MLP, HGNN, combination of MLP+HGNN with different hyperparameters˙ And we have the 2-class or 3-class (4 different cut-offs) classification, where we provide the accuracy and the time difference regression, where we show the MSE.
For all of those variants we provide one table with the train-test split results. Afterwards we provide the results of the cross validation, which was done for the models k-NN, decision tree, random forest, SVM.

Then, the three best models based on the accuracy for all DBMS and features settings are given for the 3-class classifications with cut-offs 0.1, 0.05 and 0.01.

Followed by the feature importances for all features, which are calculated with the Gini importance for the final model, which is the decision tree, for all data versions.

In the end, the final decision tree models are visualized.

A.1 PostgreSQL: Basic features

Model	Acc
5-NN	0.7966
Decision tree	0.8241
Random forest	0.8276
SVM linear	0.7000
SVM poly	0.6690
SVM rbf	0.7103
MLP: 17-5-2	0.7517
MLP: 17-10-2	0.7207
MLP: 17-20-2	0.7517
MLP: 17-25-2	0.7517
MLP: 17-40-2	0.7690
MLP: 17-60-2	0.7828
MLP: 17-10-5-2	0.7276
MLP: 17-20-10-2	0.7552
MLP: 17-40-20-2	0.7897
MLP: 17-40-10-2	0.5621
MLP: 17-60-40-2	0.7966
MLP: 17-60-20-2	0.7655
MLP: 17-80-50-2	0.7862
MLP, small-median: 7-60-40-2	0.7000
MLP, small-mean: 7-60-40-2	0.6759
MLP, small-min: 7-60-40-2	0.6966
MLP, small-max: 7-60-40-2	0.7897
MLP, small-q25: 7-60-40-2	0.7103
MLP, small-q75: 7-60-40-2	0.7069
MLP, custom: 7-60-40-2	0.7897
HGNN: 1-16-32-2	0.8138
HGNN: 1-32-16-2	0.8103
HGNN: 1-16-32-16-2	0.8000
HGNN: 1-32-64-2	0.8034
HGNN: 1-4-16-2	0.8103
combi: 17-60-40-2/1-16-32-2/4-2	0.8103
combi: 17-60-40-5/1-16-32-5/10-2	0.8138
combi: 17-60-40-5/1-16-32-5/10-20-2	0.8483
combi: 17-60-40-10/1-16-32-10/20-40-2	0.8621
combi: 17-60-40-10/1-16-32-10/20-60-20-2	0.8517

Table A.1: Accuracy of ML models for 2 classes for PostgreSQL with basic features (train-test split).

Model	Acc(0.5)	Acc(0.1)	Acc(0.05)	Acc(0.01)
5-NN	0.8345	0.7448	0.7138	0.7414
Decision tree	0.8621	0.7759	0.7345	0.7621
Random forest	0.8448	0.7793	0.731	0.7724
SVM linear	0.731	0.6759	0.6034	0.6552
SVM poly	0.7138	0.6172	0.6172	0.6655
SVM rbf	0.7103	0.6138	0.6517	0.669
MLP 17-5-3	0.6172	0.469	0.5448	0.669
MLP 17-10-3	0.7172	0.5448	0.6	0.6655
MLP 17-20-3	0.7276	0.6759	0.6621	0.7241
MLP 17-25-3	0.7793	0.6931	0.6759	0.6621
MLP 17-40-3	0.7828	0.6655	0.6586	0.7172
MLP 17-60-3	0.7793	0.6793	0.6966	0.7241
MLP 17-10-5-3	0.6897	0.4862	0.5483	0.6862
MLP 17-20-10-3	0.7759	0.7276	0.6655	0.7276
MLP 17-40-20-3	0.7862	0.7448	0.6517	0.7241
MLP 17-40-10-3	0.7828	0.731	0.5793	0.6724
MLP 17-60-40-3	0.7897	0.7483	0.6966	0.7172
MLP 17-60-20-3	0.7828	0.7379	0.7138	0.7
MLP 17-80-50-3	0.7862	0.7276	0.7138	0.7276
MLP small-median: 7-60-40-3	0.7655	0.6897	0.6586	0.669
MLP small-mean: 7-60-40-3	0.7897	0.7069	0.6483	0.6483
MLP small-min: 7-60-40-3	0.7724	0.7241	0.6276	0.6345
MLP small-max: 7-60-40-3	0.7897	0.7069	0.7069	0.7379
MLP small-q25: 7-60-40-3	0.769	0.7276	0.6448	0.6586
MLP small-q75: 7-60-40-3	0.7828	0.7034	0.6586	0.6517
MLP custom: 7-60-40-3	0.8034	0.7724	0.7276	0.7621
HGNN 1-16-32-3	0.8276	0.7724	0.7655	0.769
HGNN 1-32-16-3	0.831	0.7828	0.7517	0.7655
HGNN 1-16-32-16-3	0.8276	0.7931	0.7621	0.7517
HGNN 1-32-64-3	0.8276	0.7931	0.7655	0.7621
HGNN 1-4-16-3	0.8414	0.7966	0.7586	0.7552
combi 17-60-40-3/1-16-32-3/6-3	0.8414	0.8034	0.731	0.7759
combi 17-60-40-5/1-16-32-5/10-3	0.8655	0.8103	0.7621	0.7828
combi 17-60-40-5/1-16-32-5/10-20-3	0.9034	0.8069	0.7759	0.8069
combi 17-60-40-10/1-16-32-10/20-40-3	0.9069	0.8034	0.7621	0.8
combi 17-60-40-10/1-16-32-10/20-60-20-3	0.9	0.8034	0.7586	0.7966

Table A.2: Accuracy of ML models for 3 classes with the different cut-offs for PostgreSQL with basic features (train-test split).

Model	MSE
5-NN	0.585
Decision tree	0.5809
Random forest	0.556
SVM linear	1.7778
SVM poly	1.7955
SVM rbf	1.6447
MLP: 17-5-1	1.3807
MLP: 17-10-1	1.4159
MLP: 17-20-1	1.3757
MLP: 17-25-1	1.3631
MLP: 17-40-1	1.3528
MLP: 17-60-1	1.3012
MLP: 17-10-5-1	1.2953
MLP: 17-20-10-1	1.2512
MLP: 17-40-20-1	1.2136
MLP: 17-40-10-1	1.2511
MLP: 17-60-40-1	1.1265
MLP: 17-60-20-1	1.2061
MLP: 17-80-50-1	1.1359
MLP, small-median: 7-60-40-1	1.449
MLP, small-mean: 7-60-40-1	1.3585
MLP, small-min: 7-60-40-1	1.4032
MLP, small-max: 7-60-40-1	1.2271
MLP, small-q25: 7-60-40-1	1.4469
MLP, small-q75: 7-60-40-1	1.3491
MLP, custom: 7-60-40-1	1.1578
HGNN: 1-16-32-1	0.8234
HGNN: 1-32-16-1	0.862
HGNN: 1-16-32-16-1	0.8345
HGNN: 1-32-64-1	0.8142
HGNN: 1-4-16-1	0.8933
combi: 17-60-40-2/1-32-64-2/4-1	0.78
combi: 17-60-40-5/1-32-64-5/10-1	0.7666
combi: 17-60-40-5/1-32-64-5/10-20-1	0.6833
combi: 17-60-40-10/1-32-64-10/20-40-1	0.5856
combi: 17-60-40-10/1-32-64-10/20-60-20-1	0.6444

Table A.3: Mean Squared Error of ML models for time difference for PostgreSQL with basic features (train-test split).

A.2 PostgreSQL: Basic features + POS features

Model	Acc
5-NN	0.8862
Decision tree	0.9276
Random forest	0.9448
SVM linear	0.8379
SVM poly	0.8517
SVM rbf	0.8621
MLP: 30-5-2	0.8345
MLP: 30-10-2	0.8552
MLP: 30-20-2	0.8621
MLP: 30-25-2	0.869
MLP: 30-40-2	0.8655
MLP: 30-60-2	0.8759
MLP: 30-10-5-2	0.8483
MLP: 30-20-10-2	0.8655
MLP: 30-40-20-2	0.8759
MLP: 30-40-10-2	0.8828
MLP: 30-60-40-2	0.9172
MLP: 30-60-20-2	0.8931
MLP: 30-80-50-2	0.8862
MLP, small-median: 10-60-40-2	0.8862
MLP, small-mean: 10-60-40-2	0.8586
MLP, small-min: 10-60-40-2	0.869
MLP, small-max: 10-60-40-2	0.8897
MLP, small-q25: 10-60-40-2	0.8897
MLP, small-q75: 10-60-40-2	0.8759
MLP, custom: 10-60-40-2	0.9069
HGNN: 1-16-32-2	0.8138
HGNN: 1-32-16-2	0.8103
HGNN: 1-16-32-16-2	0.8
HGNN: 1-32-64-2	0.8034
HGNN: 1-4-16-2	0.8103
combi: 30-60-40-2/1-16-32-2/4-2	0.7138
combi: 30-60-40-5/1-16-32-5/10-2	0.8034
combi: 30-60-40-5/1-16-32-5/10-20-2	0.5966
combi: 30-60-40-10/1-16-32-10/20-40-2	0.6379
combi: 30-60-40-10/1-16-32-10/20-60-20-2	0.8276

Table A.4: Accuracy of ML models for 2 classes for PostgreSQL with basic features+POS features (train-test split).

Model	Acc(0.5)	Acc(0.1)	Acc(0.05)	Acc(0.01)
5-NN 0.5	0.8862	0.8448	0.8069	0.8414
Decision tree 0.5	0.9448	0.8897	0.8897	0.8724
Random forest 0.5	0.9552	0.9069	0.8931	0.8862
SVM linear 0.5	0.8552	0.8241	0.7828	0.8000
SVM poly 0.5	0.8621	0.8172	0.8000	0.8138
SVM rbf 0.5	0.8586	0.8000	0.8000	0.8241
MLP 0.5: 30-5-3	0.8207	0.7138	0.7345	0.7966
MLP 0.5: 30-10-3	0.5828	0.7172	0.7345	0.8172
MLP 0.5: 30-20-3	0.8621	0.8241	0.8103	0.8172
MLP 0.5: 30-25-3	0.8759	0.8241	0.7931	0.8241
MLP 0.5: 30-40-3	0.8828	0.8586	0.8000	0.8241
MLP 0.5: 30-60-3	0.8862	0.8448	0.8103	0.8207
MLP 0.5: 30-10-5-3	0.5828	0.8069	0.8034	0.8207
MLP 0.5: 30-20-10-3	0.8103	0.7345	0.7966	0.8172
MLP 0.5: 30-40-20-3	0.8690	0.8172	0.8034	0.8103
MLP 0.5: 30-40-10-3	0.5828	0.8000	0.8103	0.5241
MLP 0.5: 30-60-40-3	0.8897	0.8793	0.8276	0.8172
MLP 0.5: 30-60-20-3	0.8621	0.8379	0.8172	0.8241
MLP 0.5: 30-80-50-3	0.8724	0.8414	0.8276	0.8207
MLP 0.5, small-median: 10-60-40-3	0.8655	0.8379	0.7793	0.8069
MLP 0.5, small-mean: 10-60-40-3	0.8621	0.8414	0.7966	0.8138
MLP 0.5, small-min: 10-60-40-3	0.8621	0.8276	0.7897	0.7931
MLP 0.5, small-max: 10-60-40-3	0.8966	0.8379	0.7897	0.8379
MLP 0.5, small-q25: 10-60-40-3	0.8759	0.8241	0.7897	0.8069
MLP 0.5, small-q75: 10-60-40-3	0.8621	0.8621	0.8138	0.8138
MLP 0.5, custom: 10-60-40-3	0.9138	0.8690	0.8414	0.8517
HGNN 0.5: 1-16-32-3	0.8276	0.7724	0.7655	0.7690
HGNN 0.5: 1-32-16-3	0.8310	0.7828	0.7517	0.7655
HGNN 0.5: 1-16-32-16-3	0.8276	0.7931	0.7621	0.7517
HGNN 0.5: 1-32-64-3	0.8276	0.7931	0.7655	0.7621
HGNN 0.5: 1-4-16-3	0.8414	0.7966	0.7586	0.7552
combi 0.5: 30-60-40-3/1-4-16-3/6-3	0.7828	0.7172	0.7414	0.7310
combi 0.5: 30-60-40-5/1-4-16-5/10-3	0.7000	0.7517	0.6897	0.7379
combi 0.5: 30-60-40-5/1-4-16-5/10-20-3	0.6379	0.7724	0.5034	0.5552
combi 0.5: 30-60-40-10/1-4-16-10/20-40-3	0.7414	0.4621	0.5241	0.4655
combi 0.5: 30-60-40-10/1-4-16-10/20-60-20-3	0.5759	0.4621	0.5138	0.5310

Table A.5: Accuracy of ML models for 3 classes with the different cut-offs for PostgreSQL with basic features+POS features (train-test split).

Model	MSE
5-NN	0.2581
Decision tree	0.0846
Random forest	0.0661
SVM linear	1.1992
SVM poly	1.0291
SVM rbf	0.9154
MLP: 30-5-1	0.9773
MLP: 30-10-1	1.0136
MLP: 30-20-1	0.7443
MLP: 30-25-1	0.7545
MLP: 30-40-1	0.7207
MLP: 30-60-1	0.7173
MLP: 30-10-5-1	0.9218
MLP: 30-20-10-1	0.6871
MLP: 30-40-20-1	0.5812
MLP: 30-40-10-1	0.7146
MLP: 30-60-40-1	0.4746
MLP: 30-60-20-1	0.4783
MLP: 30-80-50-1	0.4509
MLP, small-median: 10-80-50-1	0.7646
MLP, small-mean: 10-80-50-1	0.6805
MLP, small-min: 10-80-50-1	0.7587
MLP, small-max: 10-80-50-1	0.7327
MLP, small-q25: 10-80-50-1	0.6975
MLP, small-q75: 10-80-50-1	0.6023
MLP, custom: 10-80-50-1	0.5671
HGNN: 1-16-32-1	0.8234
HGNN: 1-32-16-1	0.862
HGNN: 1-16-32-16-1	0.8345
HGNN: 1-32-64-1	0.8142
HGNN: 1-4-16-1	0.8933

Table A.6: Mean Squared Error of ML models for time difference for PostgreSQL with basic features+POS features (train-test split).

A.3 DuckDB data: Basic features

Model	Acc
5-NN	0.8182
Decision tree	0.8322
Random forest	0.8357
SVM linear	0.6678
SVM poly	0.7168
SVM rbf	0.7203
MLP: 17-5-2	0.7308
MLP: 17-10-2	0.6329
MLP: 17-20-2	0.7657
MLP: 17-25-2	0.7552
MLP: 17-40-2	0.7448
MLP: 17-60-2	0.7552
MLP: 17-10-5-2	0.7063
MLP: 17-20-10-2	0.7098
MLP: 17-40-20-2	0.7762
MLP: 17-40-10-2	0.7413
MLP: 17-60-40-2	0.7378
MLP: 17-60-20-2	0.7378
MLP: 17-80-50-2	0.7832
MLP, small-median: 7-80-50-2	0.7133
MLP, small-mean: 7-80-50-2	0.7063
MLP, small-min: 7-80-50-2	0.7063
MLP, small-max: 7-80-50-2	0.7587
MLP, small-q25: 7-80-50-2	0.6853
MLP, small-q75: 7-80-50-2	0.7028
MLP, custom: 7-80-50-2	0.7727
HGNN: 1-16-32-2	0.8392
HGNN: 1-32-16-2	0.8182
HGNN: 1-16-32-16-2	0.8357
HGNN: 1-32-64-2	0.8427
HGNN: 1-4-16-2	0.8217
combi: 17-80-40-2/1-32-64-2/4-2	0.8427
combi: 17-80-50-5/1-32-64-5/10-2	0.8427
combi: 17-80-50-5/1-32-64-5/10-20-2	0.8636
combi: 17-80-50-10/1-32-64-10/20-40-2	0.8566
combi: 17-80-50-10/1-32-64-10/20-60-20-2	0.8531

Table A.7: Accuracy of ML models for 2 classes for DuckDB with basic features (train-test split).

Model	Acc(0.5)	Acc(0.1)	Acc(0.05)	Acc(0.01)
5-NN	0.7727	0.6469	0.6923	0.8112
Decision tree	0.8077	0.7063	0.7343	0.8322
Random forest	0.8147	0.6958	0.7273	0.8322
SVM linear	0.7203	0.5245	0.5874	0.6748
SVM poly	0.7273	0.5140	0.6399	0.7098
SVM rbf	0.7413	0.5105	0.6329	0.7063
MLP 17-5-3	0.6469	0.3916	0.4580	0.6783
MLP 17-10-3	0.6958	0.5000	0.5035	0.6434
MLP 17-20-3	0.7413	0.5490	0.6329	0.7168
MLP 17-25-3	0.7378	0.5629	0.6049	0.7028
MLP 17-40-3	0.7483	0.5490	0.6434	0.7308
MLP 17-60-3	0.7413	0.6049	0.6294	0.7308
MLP 17-10-5-3	0.6364	0.4895	0.5105	0.6818
MLP 17-20-10-3	0.7727	0.5664	0.6469	0.7413
MLP 17-40-20-3	0.7168	0.5664	0.6329	0.7448
MLP 17-40-10-3	0.7657	0.5035	0.6818	0.7413
MLP 17-60-40-3	0.7657	0.6329	0.6713	0.7483
MLP 17-60-20-3	0.7902	0.6329	0.6434	0.7483
MLP 17-80-50-3	0.7797	0.6329	0.6923	0.7587
MLP small-median	0.7692	0.5524	0.6329	0.7203
MLP small-mean	0.7483	0.5455	0.6294	0.7063
MLP small-min	0.7692	0.5490	0.5839	0.7028
MLP small-max	0.7937	0.6364	0.6538	0.7657
MLP small-q25	0.7657	0.5664	0.6434	0.6923
MLP small-q75	0.7587	0.5594	0.6154	0.7378
MLP custom	0.7832	0.6259	0.6993	0.7483
HGNN 1-16-32-3	0.8566	0.7483	0.7308	0.8287
HGNN 1-32-16-3	0.8357	0.7343	0.7308	0.8287
HGNN 1-16-32-16-3	0.8427	0.7168	0.7203	0.8322
HGNN 1-32-64-3	0.8287	0.7238	0.7133	0.8322
HGNN 1-4-16-3	0.8462	0.7413	0.7343	0.8287
combi 17-80-50-3/1-16-32-3/6-3	0.8392	0.7168	0.7413	0.8252
combi 17-80-50-5/1-16-32-5/10-3	0.8287	0.7308	0.7378	0.8217
combi 17-80-50-5/1-16-32-5/10-20-3	0.8776	0.7378	0.7727	0.8392
combi 17-80-50-10/1-16-32-10/20-40-3	0.8846	0.7168	0.7448	0.8601
combi 17-80-50-10/1-16-32-10/20-60-20-3	0.8811	0.7168	0.7448	0.8357

Table A.8: Accuracy of ML models for 3 classes with the different cut-offs for DuckDB with basic features (train-test split).

Model	MSE
5-NN	0.5873
Decision tree	0.4905
Random forest	0.49
SVM linear	1.6966
SVM poly	1.7413
SVM rbf	1.6184
MLP: 17-5-1	1.5051
MLP: 17-10-1	1.5547
MLP: 17-20-1	1.4512
MLP: 17-25-1	1.4722
MLP: 17-40-1	1.4442
MLP: 17-60-1	1.4115
MLP: 17-10-5-1	1.4004
MLP: 17-20-10-1	1.3451
MLP: 17-40-20-1	1.2638
MLP: 17-40-10-1	1.3881
MLP: 17-60-40-1	1.0096
MLP: 17-60-20-1	1.1247
MLP: 17-80-50-1	1.0275
MLP, small-median: 7-60-40-1	1.431
MLP, small-mean: 7-60-40-1	1.4032
MLP, small-min: 7-60-40-1	1.5349
MLP, small-max: 7-60-40-1	1.253
MLP, small-q25: 7-60-40-1	1.4586
MLP, small-q75: 7-60-40-1	1.3997
MLP, custom: 7-60-40-1	1.134
HGNN: 1-16-32-1	0.611
HGNN: 1-32-16-1	0.6607
HGNN: 1-16-32-16-1	0.5992
HGNN: 1-32-64-1	0.5969
HGNN: 1-4-16-1	0.614
combi: 17-60-40-2/1-32-64-2/4-1	0.5352
combi: 17-60-40-5/1-32-64-5/10-1	0.542
combi: 17-60-40-5/1-32-64-5/10-20-1	0.3861
combi: 17-60-40-10/1-32-64-10/20-40-1	0.3796
combi: 17-60-40-10/1-32-64-10/20-60-20-1	0.3778

Table A.9: Mean Squared Error of ML models for time difference for DuckDB with basic features (train-test split).

A.4 DuckDB data: Basic features + DuckDB features

Model	Acc
5-NN	0.8601
Decision tree	0.8706
Random forest	0.8776
SVM linear	0.7308
SVM poly	0.6958
SVM rbf	0.7028
MLP: 23-5-2	0.7308
MLP: 23-10-2	0.5385
MLP: 23-20-2	0.7657
MLP: 23-25-2	0.7832
MLP: 23-40-2	0.7937
MLP: 23-60-2	0.7867
MLP: 23-10-5-2	0.5385
MLP: 23-20-10-2	0.7133
MLP: 23-40-20-2	0.7517
MLP: 23-40-10-2	0.7937
MLP: 23-60-40-2	0.7552
MLP: 23-60-20-2	0.8077
MLP: 23-80-50-2	0.8042
MLP, small-median: 8-60-20-2	0.7203
MLP, small-mean: 8-60-20-2	0.7378
MLP, small-min: 8-60-20-2	0.7413
MLP, small-max: 8-60-20-2	0.7692
MLP, small-q25: 8-60-20-2	0.7133
MLP, small-q75: 8-60-20-2	0.6678
MLP, custom: 8-60-20-2	0.8566
HGNN: 1-16-32-2	0.8601
HGNN: 1-32-16-2	0.8357
HGNN: 1-16-32-16-2	0.8741
HGNN: 1-32-64-2	0.8462
HGNN: 1-4-16-2	0.8671
combi: 23-60-20-2/1-16-32-16-2/4-2	0.8392
combi: 23-60-20-5/1-16-32-16-5/10-2	0.5629
combi: 23-60-20-5/1-16-32-16-5/10-20-2	0.8601
combi: 23-60-20-10/1-16-32-16-10/20-40-2	0.8636
combi: 23-60-20-10/1-16-32-16-10/20-60-20-2	0.5385

Table A.10: Accuracy of ML models for 2 classes for DuckDB with basic features+DDB features (train-test split).

Model	Acc(0.5)	Acc(0.1)	Acc(0.05)	Acc(0.01)
5-NN	0.8776	0.7587	0.7832	0.8462
Decision tree	0.9021	0.7727	0.8147	0.8706
Random forest	0.8986	0.7727	0.8077	0.8636
SVM linear	0.7692	0.6364	0.6853	0.7203
SVM poly	0.8182	0.6119	0.6503	0.6993
SVM rbf	0.7867	0.5734	0.6224	0.6993
MLP 23-5-3	0.5455	0.3916	0.4790	0.5315
MLP 23-10-3	0.7762	0.4545	0.5420	0.6783
MLP 23-20-3	0.7797	0.6643	0.5175	0.6958
MLP 23-25-3	0.7832	0.6748	0.6853	0.7622
MLP 23-40-3	0.7762	0.6678	0.6678	0.7483
MLP 23-60-3	0.8217	0.6364	0.7063	0.7552
MLP 23-10-5-3	0.5455	0.3916	0.6259	0.7203
MLP 23-20-10-3	0.8007	0.5559	0.5804	0.7098
MLP 23-40-20-3	0.8112	0.6503	0.6643	0.7483
MLP 23-40-10-3	0.6643	0.4790	0.4790	0.5315
MLP 23-60-40-3	0.8497	0.6958	0.7552	0.7587
MLP 23-60-20-3	0.8252	0.3916	0.7657	0.7692
MLP 23-80-50-3	0.8462	0.6748	0.7133	0.7587
MLP 23-60-40-3 small-median	0.8007	0.5839	0.6399	0.6783
MLP 23-60-40-3 small-mean	0.8287	0.6224	0.6434	0.6748
MLP 23-60-40-3 small-min	0.8077	0.6224	0.6573	0.7063
MLP 23-60-40-3 small-max	0.8217	0.6783	0.7063	0.7797
MLP 23-60-40-3 small-q25	0.8182	0.6014	0.6538	0.6783
MLP 23-60-40-3 small-q75	0.8147	0.6119	0.6538	0.6538
MLP 23-60-40-3 custom	0.8636	0.7587	0.7937	0.8252
HGNN 1-16-32-3	0.8252	0.6958	0.7552	0.8531
HGNN 1-32-16-3	0.8322	0.7028	0.7378	0.8636
HGNN 1-16-32-16-3	0.8287	0.7063	0.7727	0.8392
HGNN 1-32-64-3	0.8392	0.7168	0.7692	0.8566
HGNN 1-4-16-3	0.8357	0.6888	0.7343	0.8531
combi 23-60-40-3/1-32-64-3/6-3	0.8322	0.7063	0.7483	0.8497
combi 23-60-40-5/1-32-64-5/10-3	0.8357	0.7063	0.7552	0.8531
combi 23-60-40-5/1-32-64-5/10-20-3	0.8392	0.4056	0.5175	0.5385
combi 23-60-40-10/1-32-64-10/20-40-3	0.5909	0.7063	0.7343	0.5594
combi 23-60-40-10/1-32-64-10/20-60-20-3	0.5664	0.3916	0.4790	0.5315

Table A.11: Accuracy of ML models for 3 classes with the different cut-offs for DuckDB with basic features+DDB features (train-test split).

Model	MSE
5-NN	0.3076
Decision tree	0.2086
Random forest	0.2055
SVM linear	1.2302
SVM poly	1.0955
SVM rbf	1.3344
MLP: 23-5-1	1.81
MLP: 23-10-1	1.1918
MLP: 23-20-1	0.9993
MLP: 23-25-1	1.0652
MLP: 23-40-1	0.8425
MLP: 23-60-1	0.7481
MLP: 23-10-5-1	1.4743
MLP: 23-20-10-1	0.7798
MLP: 23-40-20-1	0.6151
MLP: 23-40-10-1	0.6515
MLP: 23-60-40-1	0.5078
MLP: 23-60-20-1	0.5614
MLP: 23-80-50-1	0.4525
MLP, small-median: 8-80-50-1	0.9157
MLP, small-mean: 8-80-50-1	0.9215
MLP, small-min: 8-80-50-1	0.7122
MLP, small-max: 8-80-50-1	0.9088
MLP, small-q25: 8-80-50-1	0.7906
MLP, small-q75: 8-80-50-1	0.9373
MLP, custom: 8-80-50-1	0.5016
HGNN: 1-16-32-1	0.4651
HGNN: 1-32-16-1	0.4756
HGNN: 1-16-32-16-1	0.4681
HGNN: 1-32-64-1	0.4625
HGNN: 1-4-16-1	0.4869
combi: 23-80-50-2/1-32-64-2/4-1	0.513
combi: 23-80-50-5/1-32-64-5/10-1	0.4793
combi: 23-80-50-5/1-32-64-5/10-20-1	0.5043
combi: 23-80-50-10/1-32-64-10/20-40-1	0.4777
combi: 23-80-50-10/1-32-64-10/20-60-20-1	3.1948

Table A.12: Mean Squared Error of ML models for time difference for DuckDB with basic features+DDB features (train-test split).

A.5 SparkSQL data: Basic features

Model	Acc
5-NN	0.8294
Decision tree	0.8362
Random forest	0.8328
SVM linear	0.6792
SVM poly	0.6962
SVM rbf	0.7133
MLP: 17-5-2	0.7201
MLP: 17-10-2	0.6928
MLP: 17-20-2	0.7167
MLP: 17-25-2	0.7031
MLP: 17-40-2	0.6962
MLP: 17-60-2	0.7133
MLP: 17-10-5-2	0.7065
MLP: 17-20-10-2	0.7747
MLP: 17-40-20-2	0.7543
MLP: 17-40-10-2	0.7065
MLP: 17-60-40-2	0.7679
MLP: 17-60-20-2	0.7304
MLP: 17-80-50-2	0.7747
MLP, small-median: 7-80-50-2	0.7679
MLP, small-mean: 7-80-50-2	0.744
MLP, small-min: 7-80-50-2	0.7372
MLP, small-max: 7-80-50-2	0.744
MLP, small-q25: 7-80-50-2	0.7509
MLP, small-q75: 7-80-50-2	0.7372
MLP, custom: 7-80-50-2	0.7679
HGNN: 1-16-32-2	0.8055
HGNN: 1-32-16-2	0.7986
HGNN: 1-16-32-16-2	0.7986
HGNN: 1-32-64-2	0.8055
HGNN: 1-4-16-2	0.7986
combi: 17-80-50-2/1-16-32-2/4-2	0.8089
combi: 17-80-50-5/1-16-32-5/10-2	0.802
combi: 17-80-50-5/1-16-32-5/10-20-2	0.8294
combi: 17-80-50-10/1-16-32-10/20-40-2	0.8225
combi: 17-80-50-10/1-16-32-10/20-60-20-2	0.8396

Table A.13: Accuracy of ML models for 2 classes for SparkSQL with basic features (train-test split).

Model	Acc(0.5)	Acc(0.1)	Acc(0.05)	Acc(0.01)
5-NN	0.8635	0.727	0.744	0.8259
Decision tree	0.8908	0.7645	0.7816	0.8294
Random forest	0.8908	0.7645	0.785	0.8225
SVM linear	0.7884	0.5768	0.6109	0.6758
SVM poly	0.7952	0.6007	0.6519	0.6928
SVM rbf	0.7986	0.6314	0.628	0.7031
MLP: 17-5-3	0.7816	0.4096	0.4403	0.6928
MLP: 17-10-3	0.785	0.5904	0.6177	0.6894
MLP: 17-20-3	0.8123	0.5563	0.6485	0.7201
MLP: 17-25-3	0.8225	0.587	0.5939	0.6962
MLP: 17-40-3	0.8259	0.6177	0.6621	0.7133
MLP: 17-60-3	0.843	0.6007	0.6212	0.7065
MLP: 17-10-5-3	0.8191	0.5836	0.628	0.6997
MLP: 17-20-10-3	0.8123	0.6177	0.6621	0.6962
MLP: 17-40-20-3	0.8396	0.6382	0.6724	0.7167
MLP: 17-40-10-3	0.843	0.6007	0.628	0.7133
MLP: 17-60-40-3	0.8498	0.6416	0.6758	0.7099
MLP: 17-60-20-3	0.8362	0.6485	0.6689	0.7065
MLP: 17-80-50-3	0.8294	0.628	0.6962	0.7474
MLP, small-median: 7-80-50-3	0.8294	0.5802	0.6689	0.7406
MLP, small-mean: 7-80-50-3	0.8259	0.6075	0.6724	0.7474
MLP, small-min: 7-80-50-3	0.8259	0.6177	0.6689	0.7372
MLP, small-max: 7-80-50-3	0.8567	0.6689	0.7065	0.7782
MLP, small-q25: 7-80-50-3	0.8328	0.6041	0.6758	0.7304
MLP, small-q75: 7-80-50-3	0.8328	0.5973	0.6689	0.7338
MLP, custom: 7-80-50-3	0.8601	0.6724	0.6724	0.785
HGNN: 1-16-32-3	0.8771	0.6997	0.7235	0.785
HGNN: 1-32-16-3	0.8703	0.7031	0.727	0.785
HGNN: 1-16-32-16-3	0.8703	0.7065	0.7201	0.7918
HGNN: 1-32-64-3	0.884	0.6928	0.7133	0.7884
HGNN: 1-4-16-3	0.8703	0.7167	0.7406	0.7782
combi: 17-80-50-3/1-4-16-3/6-3	0.8874	0.727	0.744	0.8055
combi: 17-80-50-5/1-4-16-5/10-3	0.9113	0.727	0.7645	0.8089
combi: 17-80-50-5/1-4-16-5/10-20-3	0.9113	0.7577	0.7645	0.8259
combi: 17-80-50-10/1-4-16-10/20-40-3	0.9044	0.7577	0.7816	0.8328
combi: 17-80-50-10/1-4-16-10/20-60-20-3	0.8908	0.7406	0.7372	0.8157

Table A.14: Accuracy of ML models for 3 classes with the different cut-offs for SparkSQL with basic features (train-test split).

Model	MSE
5-NN	0.6731
Decision tree	0.579
Random forest	0.5846
SVM linear	2.3233
SVM poly	1.7783
SVM rbf	1.6123
MLP: 17-5-1	1.386
MLP: 17-10-1	1.5114
MLP: 17-20-1	1.4429
MLP: 17-25-1	1.3431
MLP: 17-40-1	1.3479
MLP: 17-60-1	1.3885
MLP: 17-10-5-1	1.3602
MLP: 17-20-10-1	1.2734
MLP: 17-40-20-1	1.2859
MLP: 17-40-10-1	1.2556
MLP: 17-60-40-1	1.1626
MLP: 17-60-20-1	1.1831
MLP: 17-80-50-1	1.1241
MLP, small-median: 7-80-50-1	1.3131
MLP, small-mean: 7-80-50-1	1.2747
MLP, small-min: 7-80-50-1	1.3441
MLP, small-max: 7-80-50-1	1.2027
MLP, small-q25: 7-80-50-1	1.34
MLP, small-q75: 7-80-50-1	1.2379
MLP, custom: 7-80-50-1	1.1544
HGNN: 1-16-32-1	0.6714
HGNN: 1-32-16-1	0.7054
HGNN: 1-16-32-16-1	0.662
HGNN: 1-32-64-1	0.6577
HGNN: 1-4-16-1	0.7163
combi: 17-80-50-2/1-32-64-2/4-1	0.5557
combi: 17-80-50-5/1-32-64-5/10-1	0.5657
combi: 17-80-50-5/1-32-64-5/10-20-1	0.4688
combi: 17-80-50-10/1-32-64-10/20-40-1	0.4346
combi: 17-80-50-10/1-32-64-10/20-60-20-1	0.4794

Table A.15: Mean Squared Error of ML models for time difference for SparkSQL with basic features (train-test split).

A.6 SparkSQL data: Basic features + POS features

Model	Acc
5-NN	0.8635
Decision tree	0.8874
Random forest	0.8908
SVM linear	0.7884
SVM poly	0.7816
SVM rbf	0.7952
MLP: 30-5-2	0.7952
MLP: 30-10-2	0.802
MLP: 30-20-2	0.8294
MLP: 30-25-2	0.7986
MLP: 30-40-2	0.8259
MLP: 30-60-2	0.8635
MLP: 30-10-5-2	0.8294
MLP: 30-20-10-2	0.8396
MLP: 30-40-20-2	0.8498
MLP: 30-40-10-2	0.8635
MLP: 30-60-40-2	0.8328
MLP: 30-60-20-2	0.8396
MLP: 30-80-50-2	0.8225
MLP, small-median: 10-40-10-2	0.8157
MLP, small-mean: 10-40-10-2	0.8055
MLP, small-min: 10-40-10-2	0.8123
MLP, small-max: 10-40-10-2	0.8328
MLP, small-q25: 10-40-10-2	0.7952
MLP, small-q75: 10-40-10-2	0.8089
MLP, custom: 10-40-10-2	0.8532
HGNN: 1-16-32-2	0.8055
HGNN: 1-32-16-2	0.7986
HGNN: 1-16-32-16-2	0.7986
HGNN: 1-32-64-2	0.8055
HGNN: 1-4-16-2	0.7986
combi: 30-40-10-2/1-16-32-2/4-2	0.7986
combi: 30-40-10-5/1-16-32-5/10-2	0.8055
combi: 30-40-10-5/1-16-32-5/10-20-2	0.7986
combi: 30-40-10-10/1-16-32-10/20-40-2	0.8157
combi: 30-40-10-10/1-16-32-10/20-60-20-2	0.686

Table A.16: Accuracy of ML models for 2 classes for SparkSQL with basic features+POS features (train-test split).

Model	Acc(0.5)	Acc(0.1)	Acc(0.05)	Acc(0.01)
5-NN	0.942	0.7782	0.7816	0.843
Decision tree	0.9693	0.7986	0.8191	0.8635
Random forest	0.9727	0.8191	0.8362	0.8703
SVM linear	0.8703	0.6724	0.7338	0.7816
SVM poly	0.8874	0.686	0.7133	0.7747
SVM rbf	0.884	0.686	0.7167	0.785
MLP: 30-5-3	0.6177	0.6689	0.7065	0.7986
MLP: 30-10-3	0.8771	0.6928	0.7304	0.8225
MLP: 30-20-3	0.8976	0.6689	0.7372	0.7952
MLP: 30-25-3	0.9317	0.7235	0.7372	0.802
MLP: 30-40-3	0.9249	0.7031	0.7406	0.8396
MLP: 30-60-3	0.9317	0.727	0.7713	0.8362
MLP: 30-10-5-3	0.8737	0.6451	0.7065	0.7747
MLP: 30-20-10-3	0.9113	0.7167	0.7474	0.8225
MLP: 30-40-20-3	0.9352	0.7133	0.7952	0.8259
MLP: 30-40-10-3	0.9215	0.4096	0.7509	0.8362
MLP: 30-60-40-3	0.9283	0.6758	0.7577	0.8089
MLP: 30-60-20-3	0.9113	0.7304	0.7474	0.8396
MLP: 30-80-50-3	0.9454	0.7338	0.7577	0.8225
MLP, small-median: 10-80-50-3	0.9454	0.7474	0.7543	0.8362
MLP, small-mean: 10-80-50-3	0.8976	0.6485	0.7372	0.8259
MLP, small-min: 10-80-50-3	0.9249	0.7509	0.7679	0.8567
MLP, small-max: 10-80-50-3	0.9215	0.4096	0.7372	0.8191
MLP, small-q25: 10-80-50-3	0.9249	0.744	0.7304	0.8157
MLP, small-q75: 10-80-50-3	0.9249	0.4096	0.785	0.7918
MLP, custom: 10-80-50-3	0.9283	0.7304	0.8089	0.8123
HGNN: 1-16-32-3	0.8771	0.6997	0.7235	0.785
HGNN: 1-32-16-3	0.8703	0.7031	0.727	0.785
HGNN: 1-16-32-16-3	0.8703	0.7065	0.7201	0.7918
HGNN: 1-32-64-3	0.884	0.6928	0.7133	0.7884
HGNN: 1-4-16-3	0.8703	0.7167	0.7406	0.7782
combi: 30-80-50-3/1-4-16-3/6-3	0.727	0.57	0.5939	0.6724
combi: 30-80-50-5/1-4-16-5/10-3	0.7201	0.5836	0.5529	0.7133
combi: 30-80-50-5/1-4-16-5/10-20-3	0.5461	0.6109	0.529	0.5563
combi: 30-80-50-10/1-4-16-10/20-40-3	0.7372	0.6758	0.5563	0.4744
combi: 30-80-50-10/1-4-16-10/20-60-20-3	0.6792	0.4778	0.471	0.6177

Table A.17: Accuracy of ML models for 3 classes with the different cut-offs for SparkSQL with basic features (train-test split).

Model	MSE
5-NN	0.2863
Decision tree	0.0222
Random forest	0.0252
SVM linear	2.1913
SVM poly	1.1065
SVM rbf	1.0176
MLP: 30-5-1	1.1316
MLP: 30-10-1	0.8828
MLP: 30-20-1	0.7482
MLP: 30-25-1	0.6789
MLP: 30-40-1	0.7076
MLP: 30-60-1	0.66
MLP: 30-10-5-1	0.9012
MLP: 30-20-10-1	0.5882
MLP: 30-40-20-1	0.5668
MLP: 30-40-10-1	0.564
MLP: 30-60-40-1	0.4182
MLP: 30-60-20-1	0.392
MLP: 30-80-50-1	0.3607
MLP, small-median: 10-80-50-1	0.6215
MLP, small-mean: 10-80-50-1	0.6637
MLP, small-min: 10-80-50-1	0.5706
MLP, small-max: 10-80-50-1	0.6081
MLP, small-q25: 10-80-50-1	0.6113
MLP, small-q75: 10-80-50-1	0.6648
MLP, custom: 10-80-50-1	0.6684
HGNN: 1-16-32-1	0.6714
HGNN: 1-32-16-1	0.7054
HGNN: 1-16-32-16-1	0.662
HGNN: 1-32-64-1	0.6577
HGNN: 1-4-16-1	0.7163

Table A.18: Mean Squared Error of ML models for time difference for SparkSQL with basic features+POS features (train-test split).

A.7 Cross-validation

	2-class	3-class				time diff
Model	**Acc**	**Acc(0.5)**	**Acc(0.1)**	**Acc(0.05)**	**Acc(0.01)**	**MSE**
5-NN	0.8175	0.8317	0.7316	0.7159	0.7765	0.7392
Decision tree	0.8282	0.8428	0.7462	0.7247	0.7857	0.6721
Random forest	0.8255	0.8405	0.7508	0.722	0.7868	0.6534
SVM linear	0.6706	0.714	0.6281	0.5801	0.6449	2.0203
SVM poly	0.6745	0.714	0.5939	0.6116	0.6714	1.9646
SVM rbf	0.6952	0.709	0.6005	0.638	0.6725	1.8261

Table A.19: Accuracy/MSE of ML models for PostgreSQL with basic features (cross validation).

	2-class	3-class				time diff
Model	**Acc**	**Acc(0.5)**	**Acc(0.1)**	**Acc(0.05)**	**Acc(0.01)**	**MSE**
5-NN	0.9072	0.913	0.8255	0.8067	0.8723	0.2713
Decision tree	0.9356	0.9463	0.857	0.8528	0.9003	0.1259
Random forest	0.9417	0.9502	0.8654	0.86	0.903	0.1078
SVM linear	0.8401	0.8512	0.7772	0.75	0.8083	1.264
SVM poly	0.8535	0.8677	0.778	0.7546	0.8209	0.9943
SVM rbf	0.8528	0.8512	0.7584	0.7515	0.8129	0.8976

Table A.20: Accuracy/MSE of ML models for PostgreSQL with basic features+POS features (cross validation).

	2-class	3-class				time diff
Model	**Acc**	**Acc(0.5)**	**Acc(0.1)**	**Acc(0.05)**	**Acc(0.01)**	**MSE**
5-NN	0.8112	0.7941	0.7032	0.737	0.796	0.6404
Decision tree	0.8275	0.8042	0.7405	0.7607	0.8139	0.5914
Random forest	0.8271	0.8073	0.7366	0.7618	0.812	0.5732
SVM linear	0.6457	0.6861	0.5614	0.5765	0.634	1.8595
SVM poly	0.6725	0.6915	0.5354	0.6018	0.6585	1.8291
SVM rbf	0.6624	0.7032	0.5287	0.5839	0.6461	1.7923

Table A.21: Accuracy/MSE of ML models for DuckDB with basic features (cross validation).

	2-class	3-class				time diff
Model	**Acc**	**Acc(0.5)**	**Acc(0.1)**	**Acc(0.05)**	**Acc(0.01)**	**MSE**
5-NN	0.8506	0.8584	0.7752	0.8001	0.8331	0.3451
Decision tree	0.8658	0.8744	0.7962	0.8195	0.8514	0.2414
Random forest	0.8662	0.8779	0.7981	0.8168	0.8487	0.2402
SVM linear	0.7009	0.7643	0.6616	0.6515	0.6951	1.3597
SVM poly	0.6923	0.783	0.6402	0.634	0.6807	1.2058
SVM rbf	0.7017	0.7573	0.6145	0.6134	0.6927	1.3968

Table A.22: Accuracy/MSE of ML models for DuckDB with basic features+DDB features (cross validation).

	2-class	3-class				time diff
Model	**Acc**	**Acc(0.5)**	**Acc(0.1)**	**Acc(0.05)**	**Acc(0.01)**	**MSE**
5-NN	0.8008	0.8447	0.7246	0.7303	0.7886	0.73
Decision tree	0.8144	0.8644	0.7295	0.742	0.8011	0.6128
Random forest	0.8129	0.8674	0.7367	0.7432	0.797	0.6137
SVM linear	0.6841	0.7659	0.553	0.5924	0.6784	2.3711
SVM poly	0.7148	0.7727	0.5917	0.6231	0.7057	1.8152
SVM rbf	0.7106	0.7701	0.6	0.6152	0.7019	1.6664

Table A.23: Accuracy/MSE of ML models for SparkSQL with basic features (cross validation).

	2-class	3-class				time diff
Model	**Acc**	**Acc(0.5)**	**Acc(0.1)**	**Acc(0.05)**	**Acc(0.01)**	**MSE**
5-NN	0.8538	0.9352	0.7943	0.7883	0.8432	0.2143
Decision tree	0.8981	0.9761	0.836	0.8333	0.8765	0.0646
Random forest	0.9027	0.9742	0.8432	0.8379	0.8792	0.0408
SVM linear	0.7932	0.8527	0.7045	0.7152	0.7848	2.0286
SVM poly	0.8004	0.8682	0.6973	0.7098	0.789	1.1621
SVM rbf	0.7992	0.8735	0.703	0.708	0.792	1.1079

Table A.24: Accuracy/MSE of ML models for SparkSQL with basic features+POS features (cross validation).

A.8 Best models for 3 classes with cut-offs 0.1, 0.05, 0.01

Data	Best	Second best	Third best
POS: basic	Combi (81.03%)	HGNN (79.66%)	Rand. Forest (77.93%)
POS: basic+POS	Rand. Forest (90.69%)	Dec. Tree (88.97%)	MLP (87.93%)
DDB: basic	HGNN (74.13%)	Combi (73.78%)	Dec. Tree (70.63%)
DDB: basic+DDB	Dec. Tree (77.27%)	Rand. Forest (77.27%)	k-NN (75.87%)
SPA: basic	Dec. Tree (76.45%)	Rand. Forest (76.45%)	Combi (75.77%)
SPA: basic+POS	Rand. Forest (81.91%)	Dec. Tree (79.86%)	k-NN (77.82%)

Table A.25: Best models for 3 classes with cut-off 0.1.

Data	Best	Second best	Third best
POS: basic	Combi (77.59%)	HGNN (76.55%)	Dec. Tree (73.45%)
POS: basic+POS	Rand. Forest (89.31%)	Dec. Tree (88.97%)	MLP (84.14%)
DDB: basic	Combi (77.27%)	Dec. Tree (73.43%)	HGNN (73.43%)
DDB: basic+DDB	Dec. Tree (81.47%)	Rand. Forest (80.77%)	MLP (79.37%)
SPA: basic	Rand. Forest (78.50%)	Dec. Tree (78.16%)	Combi (78.16%)
SPA: basic+POS	Rand. Forest (83.62%)	Dec. Tree (81.91%)	MLP (80.89%)

Table A.26: Best models for 3 classes with cut-off 0.05.

Data	Best	Second best	Third best
POS: basic	Combi (80.69%)	Rand. Forest (77.24%)	HGNN (76.90%)
POS: basic+POS	Rand. Forest (88.62%)	Dec. Tree (87.24%)	MLP (85.17%)
DDB: basic	Combi (86.01%)	Dec. Tree (83.22%)	Rand. Forest (83.22%)
DDB: basic+DDB	Dec. Tree (87.06%)	Rand. Forest (86.36%)	HGNN (86.36%)
SPA: basic	Combi (83.28%)	Dec. Tree (82.94%)	k-NN (82.59%)
SPA: basic+POS	Rand. Forest (87.03%)	Dec. Tree (86.35%)	MLP (85.67%)

Table A.27: Best models for 3 classes with cut-off 0.01.

A.9 Feature importances for final model

	POS		DDB		SPA	
	basic	basic+POS	basic	basic+DDB	basic	basic+POS
#relations	0.020445	0.018528	0.061221	**0.107746**	0.063368	0.000000
#conditions	**0.072459**	0.007986	**0.092219**	0.019112	**0.120211**	0.004042
#filters	**0.235353**	0.032789	**0.211150**	0.037105	**0.393823**	**0.157783**
#joins	0.091582	**0.000819**	**0.095617**	0.030801	0.044418	0.000000
depth	0.004309	0.009621	0.002036	0.001207	0.004182	0.006276
min(cont. counts)	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
max(cont. counts)	0.049571	0.000482	**0.060844**	**0.110673**	**0.061582**	0.024547
mean(cont. counts)	**0.488273**	0.015743	**0.408201**	**0.112311**	**0.270023**	**0.053083**
q25(cont. counts)	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
median(cont. counts)	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
q75(cont. counts)	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
min(branching f.)	0.006215	0.000970	0.005068	0.005552	0.003849	0.003574
max(branching f.)	0.002921	0.005038	0.009110	0.001885	0.003813	0.002858
mean(branching f.)	0.002310	0.003172	0.005972	0.000954	0.001634	0.001613
q25(branching f.)	0.001633	0.000917	0.009327	0.004647	0.004131	0.010581
median(branching f.)	0.001594	0.000829	0.001588	0.003648	0.010049	0.005366
q75(branching f.)	0.023335	0.002369	0.037646	0.031777	0.018918	0.008580
total cost	-	0.014370	-	-	-	**0.089817**
min(table rows)	-	0.017513	-	-	-	0.002870
max(table rows)	-	0.046585	-	-	-	0.038798
mean(table rows)	-	0.005178	-	-	-	0.034680
q25(table rows)	-	0.021162	-	-	-	0.035148
median(table rows)	-	0.021480	-	-	-	0.006461
q75(table rows)	-	0.041745	-	-	-	**0.053000**
min(join rows)	-	**0.116054**	-	-	-	0.028861
max(join rows)	-	**0.494756**	-	-	-	**0.287186**
mean(join rows)	-	0.015821	-	-	-	0.040820
q25(join rows)	-	0.017985	-	-	-	0.043857
median(join rows)	-	0.030086	-	-	-	0.026628
q75(join rows)	-	**0.058002**	-	-	-	0.033572
min(cardinality)	-	-	-	**0.107000**	-	-
max(cardinality)	-	-	-	0.022796	-	-
mean(cardinality)	-	-	-	0.084589	-	-
q25(cardinality)	-	-	-	0.022822	-	-
median(cardinality)	-	-	-	**0.241326**	-	-
q75(cardinality)	-	-	-	0.054049	-	-

Table A.28: Feature importances for the final model (=decision tree).

A.10 Visualizations of final model (decision tree)

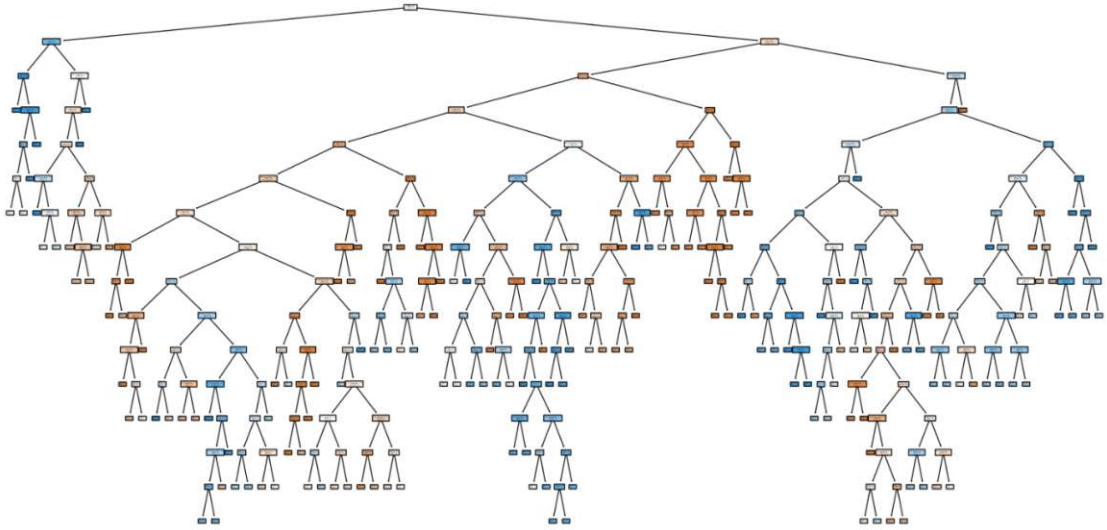


Figure A.1: Visualization of the final model (=decision tree) for PostgreSQL with basic features (train-test split).

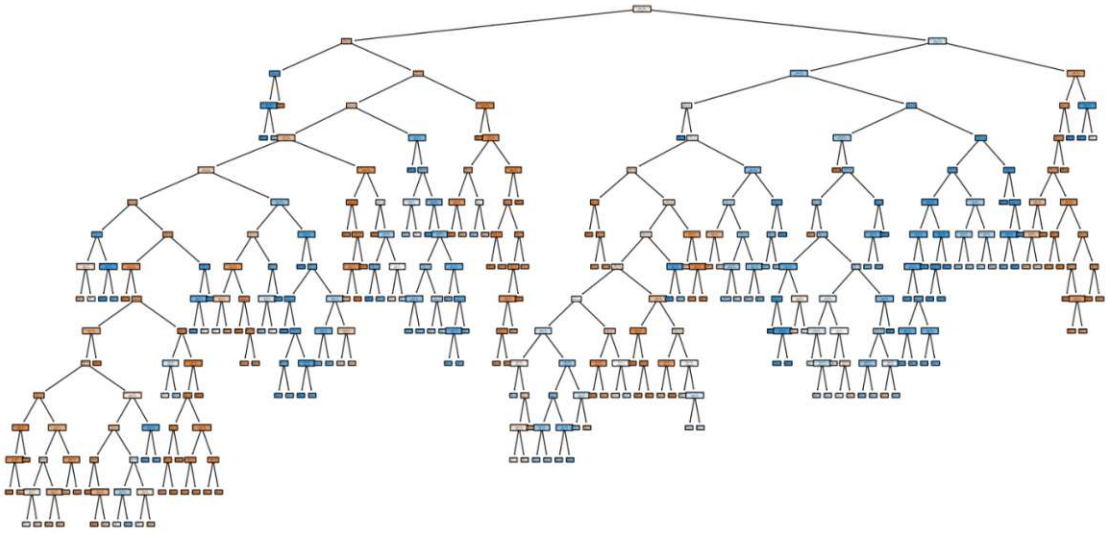


Figure A.2: Visualization of the final model (=decision tree) for DuckDB with basic features (train-test split).

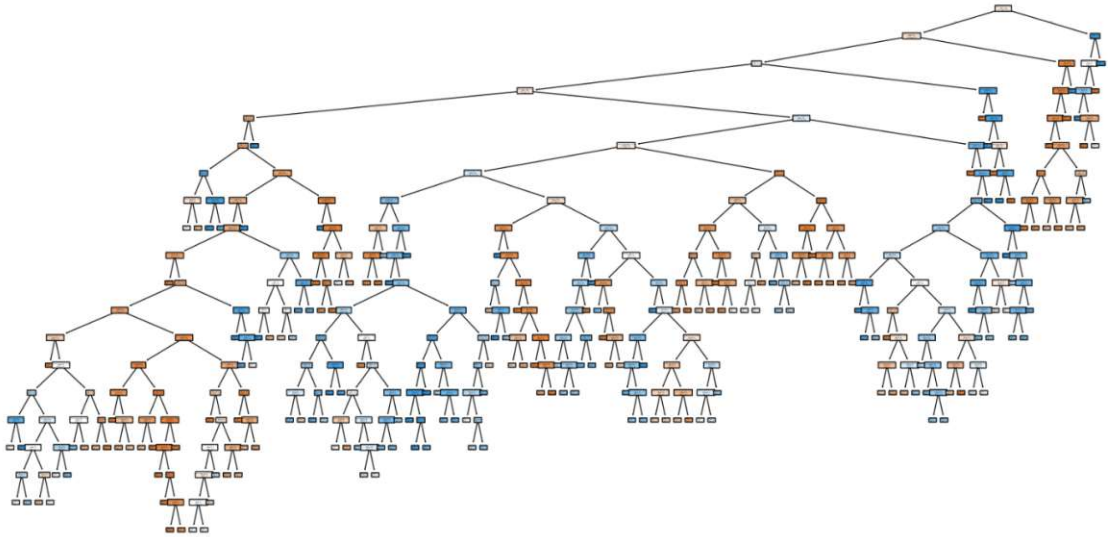


Figure A.3: Visualization of the final model (=decision tree) for DuckDB with basic features+DDB features (train-test split).

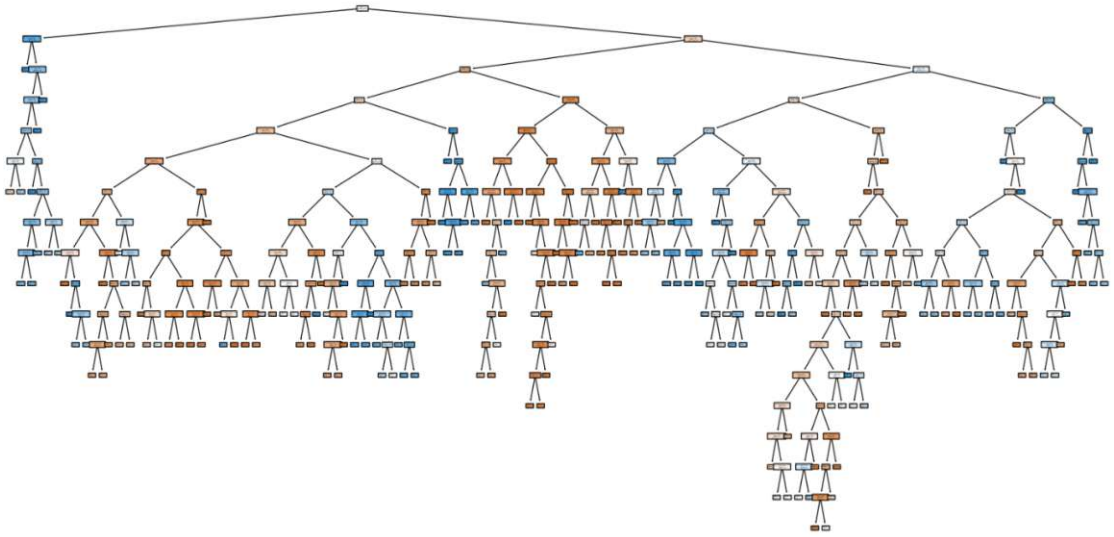


Figure A.4: Visualization of the final model (=decision tree) for SparkSQL with basic features (train-test split).

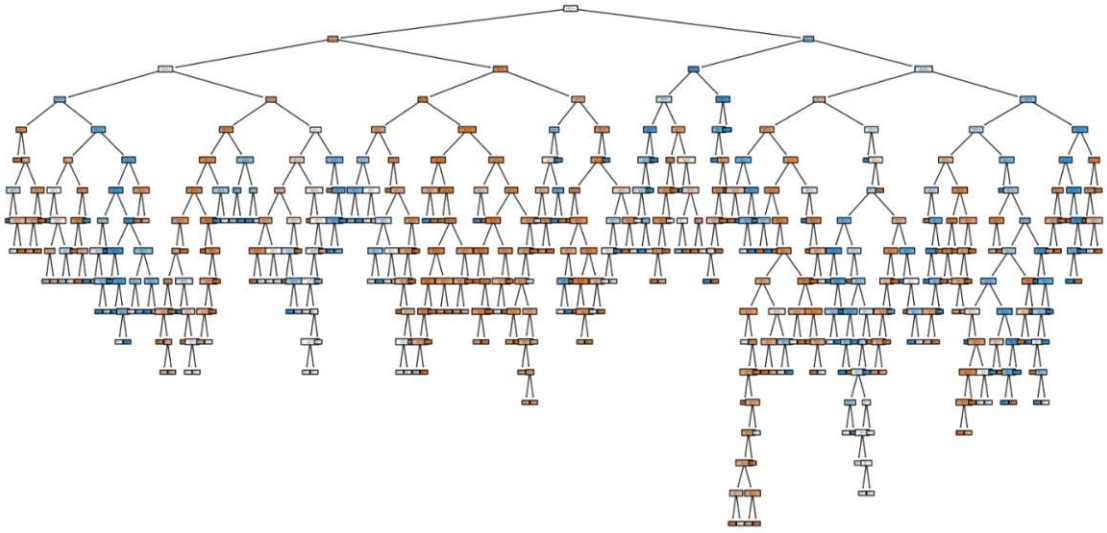


Figure A.5: Visualization of the final model (=decision tree) for SparkSQL with basic features+POS features (train-test split).

Overview of Generative AI Tools Used

As mentioned in the statement of originality I only used AI tools as support and not for the majority of the work. I used ChatGPT as support for the programming parts, for single commands and the structure of some classes (in Scala, Java, Python and Latex). Additionally, I sometimes used it to translate some words or phrases from German to English, but I wrote the whole text on my own.

List of Figures

List of Tables

List of Algorithms

Bibliography

Aberger, C. R., Lamb, A., Tu, S., Nötzli, A., Olukotun, K., & Ré, C. (2017). Emptyheaded: A relational engine for graph processing. *ACM Trans. Database Syst.*, *42*(4).

Abseher, M., Musliu, N., & Woltran, S. (2017). Improving the efficiency of dynamic programming on tree decompositions via machine learning. *Journal of Artificial Intelligence Research*, *58*, 829–858.

Alloghani, M., Al-Jumeily, D., Mustafina, J., Hussain, A., & Aljaaf, A. J. (2020). A systematic review on supervised and unsupervised machine learning algorithms for data science. In M. W. Berry, A. Mohamed, & B. W. Yap (Eds.), *Supervised and unsupervised learning for data science* (pp. 3–21). Springer International Publishing.

Almaghrebi, A., Aljuheshi, F., Rafaie, M., James, K., & Alahmad, M. (2020). Data-driven charging demand prediction at public charging stations using supervised machine learning regression methods. *Energies*, *13*, 4231.

Backer, B., Furnon, V., Shaw, P., Kilby, P., & Prosser, P. (2000). Solving vehicle routing problems using constraint programming and metaheuristics. *J. Heuristics*, *6*, 501–523.

Bagan, G., Durand, A., & Grandjean, E. (2007). On acyclic conjunctive queries and constant delay enumeration. In J. Duparc & T. A. Henzinger (Eds.), *Computer science logic* (pp. 208–222). Springer Berlin Heidelberg.

Bonifati, A., Martens, W., & Timm, T. (2017). An analytical study of large SPARQL query logs. *Proc. VLDB Endow.*, *11*(2), 149–161.

Boser, B. E., Guyon, I. M., & Vapnik, V. N. (1992). A training algorithm for optimal margin classifiers. *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, 144–152.

Brault-Baron, J. (2016). Hypergraph acyclicity revisited. *ACM Comput. Surv.*, *49*(3).

Carmeli, N., & Kröll, M. (2020). Enumeration complexity of conjunctive queries with functional dependencies. *Theory of Computing Systems*, *64*(5), 828–860.

Carmeli, N., & Kröll, M. (2021). On the enumeration complexity of unions of conjunctive queries. *ACM Trans. Database Syst.*, *46*(2).

Carmeli, N., Tziavelis, N., Gatterbauer, W., Kimelfeld, B., & Riedewald, M. (2021). Tractable orders for direct access to ranked answers of conjunctive queries. *Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, 325–341.

Carmeli, N., Zeevi, S., Berkholz, C., Conte, A., Kimelfeld, B., & Schweikardt, N. (2022). Answering (unions of) conjunctive queries using random access and random-order enumeration. *ACM Trans. Database Syst.*, *47*(3).

Caruana, R., & Niculescu-Mizil, A. (2006). An empirical comparison of supervised learning algorithms. *Proceedings of the 23rd international conference on Machine learning - ICML '06*, *2006*, 161–168.

Chandra, A. K., & Merlin, P. M. (1977). Optimal implementation of conjunctive queries in relational data bases. *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, 77–90.

Choudhary, R., & Gianey, H. K. (2017). Comprehensive review on supervised machine learning algorithms. *2017 International Conference on Machine Learning and Data Science (MLDS)*, 37–43.

Crisci, C., Ghattas, B., & Perera, G. (2012). A review of supervised machine learning algorithms and their applications to ecological data. *Ecological Modelling*, *240*, 113–122.

Cunningham, P., & Delany, S. (2007). K-nearest neighbour classifiers. *Mult Classif Syst*, *54*.

De Raedt, L., Guns, T., & Nijssen, S. (2010). Constraint programming for data mining and machine learning. *Proceedings of the National Conference on Artificial Intelligence*, *3*.

Dechter, R. (2003). *Constraint processing*. Morgan Kaufmann.

Ding, J., Minhas, U. F., Yu, J., Wang, C., Do, J., Li, Y., Zhang, H., Chandramouli, B., Gehrke, J., Kossmann, D., Lomet, D., & Kraska, T. (2020). Alex: An updatable adaptive learned index. *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 969–984.

Dutt, A., Wang, C., Nazi, A., Kandula, S., Narasayya, V., & Chaudhuri, S. (2019). Selectivity estimation for range predicates using lightweight models. *Proc. VLDB Endow.*, *12*(9), 1044–1057.

Fagin, R. (1983). Degrees of acyclicity for hypergraphs and relational database schemes. *J. ACM*, *30*(3), 514–550.

Feng, Y., You, H., Zhang, Z., Ji, R., & Gao, Y. (2019). Hypergraph neural networks. *Proceedings of the AAAI Conference on Artificial Intelligence*, *33*(01), 3558–3565.

Fischl, W., Gottlob, G., Longo, D. M., & Pichler, R. (2021). HyperBench: A benchmark and tool for hypergraphs and empirical findings. *ACM J. Exp. Algorithmics*, *26*.

Geck, G., Keppeler, J., Schwentick, T., & Spinrath, C. (2022). Rewriting with Acyclic Queries: Mind Your Head. In D. Olteanu & N. Vortmeier (Eds.), *25th international conference on database theory (icdt 2022)* (8:1–8:20, Vol. 220). Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

Geibinger, T., Mischek, F., & Musliu, N. (2019). Investigating constraint programming for real world industrial test laboratory scheduling. *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, 304–319.

Ghionna, L., Granata, L., Greco, G., & Scarcello, F. (2007). Hypertree decompositions for query optimization. *Proceedings - International Conference on Data Engineering*, 36–45.

Gottlob, G., Greco, G., Leone, N., & Scarcello, F. (2016). Hypertree decompositions: Questions and answers. *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, 57–74.

Gottlob, G., Greco, G., & Scarcello, F. (2011). Treewidth and hypertree width. *Tractability*, 3–38.

Gottlob, G., Lanzinger, M., Longo, D. M., Okulmus, C., Pichler, R., & Selzer, A. (2023). Structure-guided query evaluation: Towards bridging the gap from theory to practice. *arXiv:2303.02723*.

Gottlob, G., Leone, N., & Scarcello, F. (1998). The complexity of acyclic conjunctive queries. *Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, 706.

Gottlob, G., Leone, N., & Scarcello, F. (1999). Hypertree decompositions and tractable queries. *Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 21–32.

Graham, M. (1979). *On the universal relation* (tech. rep.). Technical report, University of Toronto.

Grohe, M., Schwentick, T., & Segoufin, L. (2001). When is the evaluation of conjunctive queries tractable? *Conference Proceedings of the Annual ACM Symposium on Theory of Computing*, 657–666.

Han, Y., Wu, Z., Wu, P., Zhu, R., Yang, J., Tan, L. W., Zeng, K., Cong, G., Qin, Y., Pfadler, A., Qian, Z., Zhou, J., Li, J., & Cui, B. (2021). Cardinality estimation in DBMS: A comprehensive benchmark evaluation. *Proc. VLDB Endow.*, *15*(4), 752–765.

Hasan, S., Thirumuruganathan, S., Augustine, J., Koudas, N., & Das, G. (2020). Deep learning models for selectivity estimation of multi-attribute queries. *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 1035–1050.

Hilprecht, B., Schmidt, A., Kulessa, M., Molina, A., Kersting, K., & Binnig, C. (2020). Deepdb: Learn from data, not from queries! *Proc. VLDB Endow.*, *13*(7), 992–1005.

Himmelstein, D. S., Lizee, A., Hessler, C., Brueggeman, L., Chen, S. L., Hadley, D., Green, A., Khankhanian, P., & Baranzini, S. E. (2017). Systematic integration of biomedical knowledge prioritizes drugs for repurposing. *eLife*, *6*, e26726.

Ho, T. K. (1995). Random decision forests. *Proceedings of 3rd International Conference on Document Analysis and Recognition*, *1*, 278–282 vol.1.

Hu, X., & Wang, Q. (2023). Computing the difference of conjunctive queries efficiently. *Proceedings of the ACM on Management of Data*, *1*, 1–26.

Idris, M., Ugarte, M., & Vansummeren, S. (2017). The dynamic yannakakis algorithm: Compact and efficient query processing under updates. *Proceedings of the 2017 ACM International Conference on Management of Data*, 1259–1274.

Idris, M., Ugarte, M., Vansummeren, S., Voigt, H., & Lehner, W. (2020). General dynamic yannakakis: Conjunctive queries with theta joins under updates. *The VLDB Journal*, *29*(2), 619–653.

Kipf, A., Kipf, T., Radke, B., Leis, V., Boncz, P., & Kemper, A. (2018). Learned cardinalities: Estimating correlated joins with deep learning. *arXiv:1809.00677*.

Kossmann, J., Halfpap, S., Jankrift, M., & Schlosser, R. (2020). Magic mirror in my hand, which is the best in the land? an experimental evaluation of index selection algorithms. *Proc. VLDB Endow.*, *13*(12), 2382–2395.

Kunjir, M., & Babu, S. (2020). Black or white? how to develop an autotuner for memory-based analytics. *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 1667–1683.

Lanzinger, M., Pichler, R., & Selzer, A. (2024). Avoiding materialisation for guarded aggregate queries. *arXiv:2406.17076*.

Leis, V., Gubichev, A., Mirchev, A., Boncz, P., Kemper, A., & Neumann, T. (2015). How good are query optimizers, really? *Proc. VLDB Endow.*, *9*(3), 204–215.

Leskovec, J., & Krevl, A. (2014). SNAP Datasets: Stanford large network dataset collection.

Lutz, C., & Przybylko, M. (2022). Efficiently enumerating answers to ontology-mediated queries. *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, 277–289.

Marcus, R., Negi, P., Mao, H., Zhang, C., Alizadeh, M., Kraska, T., Papaemmanouil, O., & Tatbul, N. (2019). Neo: A learned query optimizer. *Proc. VLDB Endow.*, *12*(11), 1705–1718.

Marcus, R., & Papaemmanouil, O. (2018). Deep reinforcement learning for join order enumeration. *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, *1*(3), 1–4.

Mhedhbi, A., Lissandrini, M., Kuiper, L., Waudby, J., & Szárnyas, G. (2021). Lsqb: A large-scale subgraph query benchmark. *GRADES-NDA '21: Proceedings of the 4th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences and Systems and Network Data Analytics*, *8*, 1–11.

Mitchell, T. (1997). *Machine learning*. McGraw-Hill Education.

Mohri, M., Rostamizadeh, A., & Talwalkar, A. (2012). *Foundations of machine learning*. MIT Press.

Muller, A., & Guido, S. (2018). *Introduction to machine learning with python: A guide for data scientists*. O'Reilly Media, Incorporated.

Naeem, S., Ali, A., Anam, S., & Ahmed, M. (2023). An unsupervised machine learning algorithms: Comprehensive review. *IJCDS Journal*, *13*, 911–921.

Nasteski, V. (2017). An overview of the supervised machine learning methods. *HORIZONS.B*, *4*, 51–62.

Nathan, V., Ding, J., Alizadeh, M., & Kraska, T. (2020). Learning multi-dimensional indexes. *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 985–1000.

Ngo, H. Q., Porat, E., Ré, C., & Rudra, A. (2012). Worst-case optimal join algorithms. *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, 37–48.

Perelman, A., & Ré, C. (2015). Duncecap: Compiling worst-case optimal query plans. *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2075–2076.

Popescu, M.-C., Balas, V., Perescu-Popescu, L., & Mastorakis, N. (2009). Multilayer perceptron and neural networks. *WSEAS Transactions on Circuits and Systems*, *8*.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, *1*(1), 81–106.

Raschka, S. (2020). Model evaluation, model selection, and algorithm selection in machine learning. *arXiv:1811.12808*.

Robertson, N., & Seymour, P. (1983). Graph minors. i. excluding a forest. *Journal of Combinatorial Theory, Series B*, *35*(1), 39–61.

Rodriguez, M., Comin, C., Casanova, D., Bruno, O., Amancio, D., Rodrigues, F., & da F. Costa, L. (2016). Clustering algorithms: A comparative approach. *PLOS ONE*, *14*.

Rumelhart, D. E., & McClelland, J. L. (1987). *Learning internal representations by error propagation*. MIT Press.

Selzer, A. (2021). Lightweight integration of query decomposition techniques into sql-based database systems.

Smola, A., & Schölkopf, B. (2004). A tutorial on support vector regression. *Statistics and Computing*, *14*, 199–222.

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT Press.

Trummer, I., Wang, J., Wei, Z., Maram, D., Moseley, S., Jo, S., Antonakakis, J., & Rayabhari, A. (2021). Skinnerdb: Regret-bounded query evaluation via reinforcement learning. *ACM Trans. Database Syst.*, *46*(3).

Tsang, E. (1993). *Foundations of constraint satisfaction*. Academic Press.

Tu, S., & Ré, C. (2015). Duncecap: Query plans using generalized hypertree decompositions. *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2077–2078.

Van Aken, D., Pavlo, A., Gordon, G. J., & Zhang, B. (2017). Automatic database management system tuning through large-scale machine learning. *Proceedings of the 2017 ACM International Conference on Management of Data*, 1009–1024.

Verhaeghe, H., Nijssen, S., Pesant, G., Quimper, C.-G., & Schaus, P. (2020). Learning optimal decision trees using constraint programming. *Constraints*, *25*, 1–25.

Volkova, S. (2024). An overview on data augmentation for machine learning. In A. Gibadullin (Ed.), *Digital and information technologies in economics and management* (pp. 143–154). Springer Nature Switzerland.

Wang, Q., Hu, X., Dai, B., & Yi, K. (2023). Change propagation without joins. *Proceedings of the VLDB Endowment*, *16*(5), 1046–1058.

Wang, Q., & Yi, K. (2022). Conjunctive queries with comparisons. *Proceedings of the 2022 International Conference on Management of Data*, 108–121.

Wang, W., Zhang, M., Chen, G., Jagadish, H. V., Ooi, B. C., & Tan, K.-L. (2016). Database meets deep learning: Challenges and opportunities. *SIGMOD Rec.*, *45*(2), 17–22.

Wang, Y., & Yi, K. (2021). Secure yannakakis: Join-aggregate queries over private data. *Proceedings of the 2021 International Conference on Management of Data*, 1969–1981.

Wilcoxon, F. (1945). Individual comparisons by ranking methods. *Biometrics Bulletin*, *1*(6), 80–83.

Wu, P., & Cong, G. (2021). A unified deep model of learning from both data and queries for cardinality estimation. *Proceedings of the 2021 International Conference on Management of Data*, 2009–2022.

Yannakakis, M. (1981). Algorithms for acyclic database schemes. *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, 82–94.

Yu, C., & Ozsoyoglu, M. (1979). An algorithm for tree-query membership of a distributed query. *COMPSAC 79. Proceedings. Computer Software and The IEEE Computer Society's Third International Applications Conference, 1979.*, 306–312.

Yu, X., Li, G., Chai, C., & Tang, N. (2020). Reinforcement learning with tree-lstm for join order selection. *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 1297–1308.

Zhang, J., Liu, Y., Zhou, K., Li, G., Xiao, Z., Cheng, B., Xing, J., Wang, Y., Cheng, T., Liu, L., Ran, M., & Li, Z. (2019). An end-to-end automatic cloud database tuning system using deep reinforcement learning. *Proceedings of the 2019 International Conference on Management of Data*, 415–432.

Zhou, X., Chai, C., Li, G., & Sun, J. (2022). Database meets artificial intelligence: A survey. *IEEE Transactions on Knowledge and Data Engineering*, *34*(3), 1096–1116.