TECHNISCHE
UNIVERSITÄT
DRESDEN

Local Learning Strategies for Data Management Components

Dissertation

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von
Dipl.-Inf. Lucas Woltmann

Gutachter:

Prof. Dr.-Ing. Wolfgang Lehner
Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur für Datenbanken
01062 Dresden

Prof. Dr. Maximilian E. Schüle
Otto-Friedrich-Universität Bamberg
Fakultät Wirtschaftsinformatik und Angewandte Informatik
Juniorprofessur für Informatik, insbesondere Data Engineering
An der Weberei 5
96047 Bamberg

Tag der Verteidigung: 14. Dezember 2023

Dresden, im Juli 2023

ABSTRACT

In a world with an ever-increasing amount of data processed, providing tools for high-quality and fast data processing is imperative. Database Management Systems (DBMSs) are complex adaptive systems supplying reliable and fast data analysis and storage capabilities. To boost the usability of DBMSs even further, a core research area of databases is performance optimization, especially for query processing.

With the successful application of Artificial Intelligence (AI) and Machine Learning (ML) in other research areas, the question arises in the database community if ML can also be beneficial for better data processing in DBMSs. This question has spawned various works successfully replacing DBMS components with ML models.

However, these *global models* have four common drawbacks due to their large, complex, and inflexible one-size-fits-all structures. These drawbacks are the high complexity of model architectures, the lower prediction quality, the slow training, and the slow forward passes. All these drawbacks stem from the core expectation to solve a certain problem with one large model at once. The full potential of ML models as DBMS components cannot be reached with a global model because the model's complexity is outmatched by the problem's complexity.

Therefore, we present a novel general strategy for using ML models to solve data management problems and to replace DBMS components. The novel strategy is based on four advantages derived from the four disadvantages of global learning strategies. In essence, our *local learning strategy* utilizes divide-and-conquer to place less complex but more expressive models specializing in sub-problems of a data management problem. It splits the problem space into less complex parts that can be solved with lightweight models. This circumvents the one-size-fits-all characteristics and drawbacks of global models. We will show that this approach and the lesser complexity of the specialized local models lead to better problem-solving qualities and DBMS performance.

The local learning strategy is applied and evaluated in three crucial use cases to replace DBMS components with ML models. These are cardinality estimation, query optimizer hinting, and integer algorithm selection. In all three applications, the benefits of the local learning strategy are demonstrated and compared to related work. We also generalize the strategy's usability for a broader application and formulate best practices with instructions for others.

CONTENTS

Acknowledgments 9

List of Abbreviations 11

1 Introduction 13

2 Problem Description 17

2.1 Preliminaries 18

2.2 Overview Machine Learning for Databases 20

2.3 Global Learning Strategies 21

2.4 Summary 23

3 The Local Learning Strategy for Cardinality Estimation 25

3.1 Preliminaries for Learned Cardinality Estimation 26

3.2 Related Work 28

3.2.1 System R 28

3.2.2 Learned Estimators with Neural Networks 29

3.2.3 Learned Estimators with Sum-Product Networks 31

3.2.4 Shortcomings of Global Strategies 32

3.3 The Local Learning Strategy for Cardinality Estimation 34

3.4 Local Model Structure 37

3.5 Query Featurization 38

3.6 Query Labeling 43

3.7 Evaluation 46

3.7.1 Setup 46

3.7.2 Model Complexity 47

3.7.3 Model Quality 48

3.7.4 Training Time and Forward Pass 49

3.7.5 Query Labeling 49

3.8 Summary 50

4 THE LOCAL LEARNING STRATEGY FOR QUERY OPTIMIZER HINTING 53

4.1 Preliminaries for Learned Query Optimizer Hinting 54

4.2 Related Work 55
4.2.1 Cardinality Estimation 56
4.2.2 Plan Enumeration 56
4.2.3 Cost Models 57
4.2.4 Direct Query Optimizer Hinting 57
4.2.5 Shortcomings of Global Strategies 58

4.3 The Local Learning Strategy For Query Optimizer Hinting 59

4.4 Smart Labeling 61

4.5 Runtime Model Maintenance 62

4.6 Evaluation 62
4.6.1 Setup 63
4.6.2 Benchmark Performance 63
4.6.3 Amount of Training Data 64
4.6.4 Contextualization 65
4.6.5 Runtime Model Maintenance 66

4.7 Summary 67

5 THE LOCAL LEARNING STRATEGY FOR INTEGER COMPRESSION SELECTION 69

5.1 Preliminaries for Learned Compression Selection 70

5.2 Related Work 71
5.2.1 Rule-based Selection 71
5.2.2 Cost-based Selection 72
5.2.3 ML-based Selection 72
5.2.4 Shortcomings of Global Strategies 73

5.3 The Local Learning Strategy For Integer Compression Selection 74

5.4 Data Generation 75

5.5 Feature Engineering 77

5.6 Models 78

5.7 Evaluation 79
5.7.1 Setup 79
5.7.2 Single Model Evaluation 81
5.7.3 Selection Strategy Evaluation 82
5.7.4 Different Hardware Setups 84

5.8 Summary 85

6 Vision and Best Practices of Generalized Local Learning Strategies 87
6.1 Problem Space Definition 88
6.2 Training Data Generation 89
6.3 Feature Engineering 89
6.4 Local Model Types 90
6.5 Quality and Performance Evaluation 90
6.6 Summary 91
7 Conclusion 93
Bibliography 97
List of Figures 105
List of Tables 107
A Appendix 109

ACKNOWLEDGMENTS

Divide et impera.
- Niccolò Machiavelli, *Il Principe*, 1532

First and foremost, I want to thank my three supervisors. Wolfgang, for the freedom to follow my ideas. Dirk for being the raised index finger of the database system community. Claudio for making sure the ML view on everything was not lost. Many thanks to my second reviewer Maximilian for reading and listening to my scientific outpourings.

Much gratitude goes to Mutti, Vati, Clemens, and my family for laying the very fundamental prerequisites for this work and reminding me that sometimes you can overtake without catching up.

Furthermore, I want to thank my colleagues, especially Ulrike, for keeping the demons of language and administration at bay. Thanks to our external partners and researchers for providing interesting topics for applied AI. Special thanks go to Katja for letting me somersault through sports research.

A special thanks to my friends Jana, Justina, Karen, and Sascha for peer-pressuring me into cooking and acknowledging a view outside the basement of Computer Science research. To Denise, Katja, and Philipp for letting me experience (and enjoy) my own adventures. To Patrick for dropping interesting takes on problems as old as humanity itself and then heading off to a Ponyhof in Brandenburg. To Chrissi and Becky for many breakfasts and much tea.

Lastly, thanks to everybody I forgot to mention, including people I met along the way, whether during my studies, my work at the university, or anywhere else.

Lucas Woltmann
Dresden, July 30th, 2023

LIST OF ABBREVIATIONS

AI Artificial intelligence

BAO Bandit Optimizer

BWH Bit width histogram

DBMS Database management system

GB Gradient boosting

GS Grouping set

IMDb Internet Movie Database

JOB Join order benchmark

LDE Limited Disjunction Encoding

ML Machine learning

MSCN Multi Set Convolutional Network

NN Neural network

OLAP Online Analytical Processing

OLTP Online Transaction Processing

PG PostgreSQL

QEP Query execution plan

RL Reinforcement learning

RPE Range Predicate Encoding

RSPN Relational Sum Product Network

SAPE Symmetric Absolute Percentage Error

SMAPE Symmetric Mean Absolute Percentage Error

SPE Singular Predicate Encoding

SPJ Select-project-join

SPN Sum Product Network

SQL Structured Query Language

TCN Tree convolutional network

UCE Universal Conjunction Encoding

1

INTRODUCTION

With the constant increase in collected data worldwide, database management systems (DBMSs) have a distinguished role in storing and processing data. Their position was described as essential for modern life in the Claremont Report in 2008 [5]. This statement was reiterated in the more recent Seattle Report from 2020 [1]. So, DBMSs have not lost any importance for society and research in the last decade. Their capability to store and analyze massive amounts of data makes them the key feature for data-based and data-intensive tasks. The occurrence of DBMSs in nearly all scenarios where there is data defines them as a ubiquitous technique. In the *data is the new oil* trend, as firstly defined by an article in The Economist [9], DBMSs are one of two essential data technologies. The other technique often mentioned with data analysis is artificial intelligence (AI) or its sub-domain machine learning (ML).

Within the area of data management, DBMSs are required to deliver answers to queries fast. This performance requirement is a core research question in the database community [1, 5]. The inclusion of new technologies to improve query performance is known as *query optimization*. Over the years, many approaches have led to a significant boost in DBMS query processing. For example, the introduction of B+ trees as indexes allows for faster row access according to a filter [7].

As the second column of the *data is the new oil* trend, ML has become one of the world's game changers and has impacted economics, politics, and sociology [6, 65]. Many day-to-day applications are enhanced by intelligent decision making with ML models. For example, with the recent introduction of ChatGPT, natural language question answering and conversational AI have gained attention within the general public [66]. Other examples include applications from the energy domain [38, 86, 96], sports [28, 92, 101], and environmental modeling [19, 59]. So, DBMS and ML both have various applications making them key technologies for data processing.

With the rise of ML in other areas and its widespread success, its desired inclusion into the DBMS world has become a subject of high interest both in academia and industry. The idea is that the combination of DBMS and ML will improve both research domains. However, there is a distinction in two different research directions. Figure 1.1 details the research gap for these two directions. The first research area is the support of ML techniques with databases. This encompasses the provisioning of data [23], the support of training [78], or the storage of models [84]. The second area is the opposite direction of the interplay between DBMS and ML. Here, ML is used to support the DBMS for performance improvements. Most of the time, ML models replace some DBMS component to either improve its quality or its performance [36, 40, 44, 47, 54]. We give more details about the component-wise ML support in DBMSs and its related research in Chapter 2.

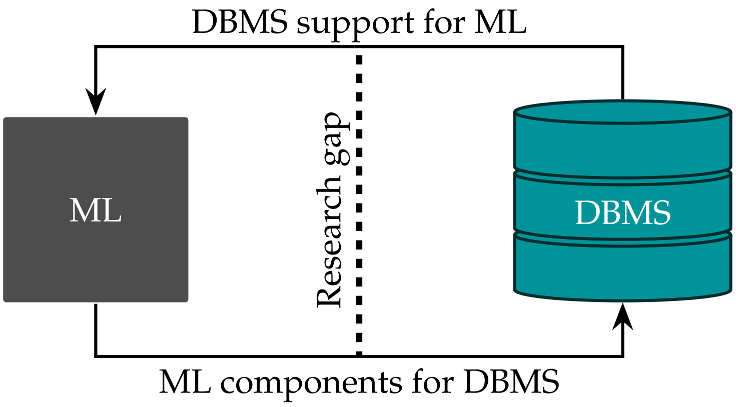


Figure 1.1: The interplay of DBMSs and ML.

In general, this work focuses on the idea of supporting the DBMS with tailored, specialized ML models for different quality and performance improvements. Most importantly, the models are not only tailored to a certain problem but follow a concept that can be applied generally.

Core contribution. The main goal of this dissertation is the different deployment of ML models as components for data management problems. As mentioned earlier, the concept of ML components in DBMSs is researched with a broad interest. However, other publications in the area of ML for DBMS train large one-size-fits-all solutions to solve the problems at hand [36, 40, 44, 54]. These solutions are called *global learning strategies* or *global models*. We will discuss these works and their main properties that make them global models. Special attention is given to the four main shortcomings of global learning strategies and we derive four corresponding advantages a competing approach should have to outperform these global models. With the four advantages in mind, we introduce a new way of solving problems with ML in DBMSs by dividing a problem space into subproblems via *divide-and-conquer*. This technique will be called a *local learning strategy* or *local models*. We introduce the local learning strategy by way of example using cardinality estimation as a challenging data management problem. We then apply the local learning strategy to two additional complex problems from the DBMS domain to show that the divide-and-conquer approach works for different problems in data management. Each of the three applications is evaluated against related state-of-the-art work regarding the four advantages generated by our local learning strategies. We will show that the general concept of a local learning strategy can improve DBMSs' processing quality and performance.

Outline. The remainder of this dissertation is structured into six chapters. Chapter 2 details the general research in the area of ML for DBMS. It will collect the general shortcomings of the global learning strategies and derive the four advantages that the local learning strategy needs to fulfill. Chapter 3 takes a closer look at the first complex challenge in DBMS: cardinality estimation. From the shortcomings of the global models in related work and the derived competing advantages, we define the *local learning strategy* for this use case. We show the general components of the strategy and evaluate its influence on quality and performance in query processing. Chapters 4 and 5 apply the concept of local models to two more use cases: query compiler hinting and integer compression selection. This will show that the local learning strategy is helpful on a manifold of challenges and offers benefits in various DBMS-related problems. For both applications, we use and extend the findings about the local learning strategy for cardinality estimation to make the local approach more comprehensive. Even though the local learning strategy needs to be adapted to each use case, we can demonstrate that the core ideas, properties, and advantages are the same over all three areas of application. To further generalize our idea, Chapter 6 details best practices and further visions for the local learning strategy and presents universal recommendations to design local learning strategies for other applications. Finally, Chapter 7 summarizes the findings of this work.

2

PROBLEM DESCRIPTION

Machine learning (ML) has become a focus for research done on data management systems. This has led to a variety of applications of ML in DBMS. Most commonly, specific parts of a DBMS can benefit from replacing its components with ML-based or learned equivalents. In this work, we want to focus on this interaction between ML and DBMS, primarily on how ML can support a DBMS and present a novel learning strategy to solve complex problems within a DBMS. To do so, we first introduce the general concepts of DBMS and ML in Section 2.1. Additionally, the interaction between the two concepts and the prerequisites for replacing DBMS components with ML models are shown. Section 2.2 gives a short overview of five domains within database research with a strong focus on ML-based components and strategies. These five areas are: indexes, join enumeration, cardinality estimation, selection strategies, and query compiler hinting. We will also detail core problems in these works that lead to limitations or shortcomings in Section 2.3. Lastly, this section also presents which advantages a possible solution to these shortcomings must offer.

2.1 PRELIMINARIES

Database systems manage data with logical connections in a *database*. In general, a database system is divided into three layers: (1) the external layers handling application access, (2) the conceptual layer handling the representation of the real-world problem in concepts and schemata, and (3) the physical layer handling the actual data storage and query processing [42, 48]. For this architecture, DBMSs use a manifold of tools to deliver the user the correct answer to a query. Therefore, a DBMS is by name and definition a *complex adaptive system* as defined by Holland [37] and Gell-Mann [32]. To be counted as such as system, a DBMS needs to fulfill two criteria: complex and adaptive. A *complex system* is a system that is made up of many interacting components. A DBMS has many components, like optimizers, buffers, or indexes, that are closely intertwined by their interactions with each other. By definition and also in real systems, these components can again contain components creating a hierarchy of components. Therefore a DBMS is a complex system. In the following, we will call the functionality of a specific (complex) component f_c. The definition for *adaptive systems* states that the behavior of such a system depends on the input and is also able to change according to different inputs. Therefore, a DBMS is an adaptive system since the user data and workload queries strongly influence its behavior. An Online Analytical Processing (OLAP) workload with a more read-focused characteristic has different demands than an Online Transaction Processing (OLTP) workload focusing more on data manipulation. A DBMS can shift its behavior according to both (or more) types of workloads.

Whereas DBMSs focus more on data storage and static analysis, ML extracts unknown information and derives universal rules from data.

$$f_m : X \mapsto Y \tag{2.1}$$

Therefore, an *ML model* learns a function f_m either by observing example data X and labels Y or by observing an environment X directly. Problems modeled from example data to labels are called *supervised*. Directly observing data without labels is named a *unsupervised* problem. Lastly, learning directly and on-the-fly from an environment and reacting to given environment states according to a reward is called *reinforcement learning* (RL). In all cases, the properties of the data set that are fixed for all data points are the *features* of the data. If the labels are from a discrete range of values, we talk about a *classification* task. The labels are then referred to as *classes*. Otherwise, with features from a continuous domain, the task is called *regression*. Most supervised tasks can be mapped to one of these two classes. After learning the function f_m in a process defined as *training*,

f_m can be applied to any new unknown data in a step called *forward pass*. Usually, the forward pass is orders of magnitude faster than the training, where a convergence of the function f_m to a minimal error must be reached.

DBMSs and ML models are powerful tools in the data community, so their interaction is manifold. More precisely, we differentiate between two concepts: DBMS for ML and ML for DBMS. DBMS for ML collects all work where the database is used to provide and preprocess data for ML models during training. This is relatively straightforward as a concept because columns in databases can directly represent features for ML models. Besides that, the training itself can be implemented as functionality in a DBMS [23]. The other concept, ML for DBMS, broadly describes the capability of ML models to support a DBMS during its adaptive lifetime. The general concept is to use learned models to replace components of DBMS as *learned components* [8, 70]. This is possible through the fitting structure of a DBMS as a hierarchical complex system. There, every component or sub-component is a candidate to be replaced with an ML model. Let us say a component solves a problem, i.e., a database-specific problem. Then the problem space P with all possible input data to be solved by the component and respective ML model is defined as

$$P : \{y = F(x) \,|\, x \in X, y \in Y\} \tag{2.2}$$

$F(x)$ is the *oracle* that always returns the right solution for a problem. The oracle is usually an abstraction for labeling the data by a human expert. Most of the time, $F(X)$ is too complex to be observed fully. A DBMS component and an ML model try to approximate F with the function f_c and f_m, respectively. The space within the whole problem space covered by any function f is called function space. Finding f_c and f_m to be as close to F as possible is an optimization problem to be solved regarding the minimization of the error E between F and f_c or f_m. Thus, F has a higher or equal *order* to f_c and f_m. Equal order in this context means the capability to solve the same problem with the same quality or the same margin of error. In other words, the order of a function is defined by how much it can cover of the given problem space P. Functions with a higher order are able to cover more of the problem space and F is the highest-order function covering the whole problem space. Figure 2.1 illustrates this assumption by depicting the problem space as a circle. On the left side, the coverage of the model function f_m (shown in green) is less than the coverage of the component function f_c (shown in orange). Neither function solves the whole problem because their coverage does not span the whole problem space. On the right side of the figure, f_m has a higher order than f_c and shows that replacing the component with the model is beneficial. Additionally, f_m completely solves the problem by having the same order and coverage as F. Therefore, replacing a component f_c with a model f_m is only feasible if f_m has a higher or equal order to f_c. Otherwise, we would lose expressiveness and are not able to solve the problem with the same level of quality.

$$f_c \preccurlyeq f_m \preccurlyeq F \tag{2.3}$$

To illustrate the previous definitions, we use the database index as an example [47]. As a component of a DBMS, the index models the function $f_c = f_i$ returning the position of a tuple within a table given a filter predicate.

$$f_c = f_i : \text{predicate} \mapsto \text{position} \tag{2.4}$$

The oracle function F delivers the correct position of tuples under any predicate on a column. Most indexes, like B-tree, can deliver the same results as F. Therefore, f_c has the same order as F: $f_c \approx F$. To replace an index with a learned model, like done by Kraska et al. [47], the function f_m needs to be of the same order as f_c and also F in this case. From this example, another optimization goal for components and ML models can be derived. If every index is capable of modeling F, how would we decide on the best one? A point answering this question in databases is enhanced quality and performance. Between two indexes, both capable of always giving the correct position, the better index returns the answer faster. This is important because one vital point in DBMS is fast workload and query execution. Therefore, components being faster is a desirable outcome. So, for an ML model to replace a component, it also should be as fast or, like in the case of indexes, faster than the component and have the same or better order than f_c.

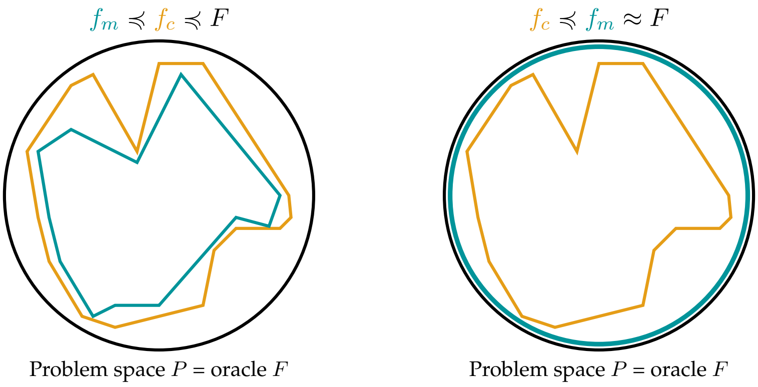

$f_m \preccurlyeq f_c \preccurlyeq F$
$f_c \preccurlyeq f_m \approx F$
Problem space P = oracle F
Problem space P = oracle F

Figure 2.1: Coverage of problem spaces by different functions f_c and f_m.

2.2 OVERVIEW MACHINE LEARNING FOR DATABASES

We now present five large domains where ML models are successfully used to replace DBMS components. This section only gives a brief overview and should show that there is intensive research within the database community for learned database components.

Indexes. Coming from the previous paragraph's example, we introduce the first area, indexes, covering access optimization in databases with ML. As mentioned before, the first breakthrough was achieved with the work by Kraska et al. [47] for learned indexes. This technique allows learning the tuple position in sorted data by showing the ML model different pairs of filter predicates and corresponding data positions. The learned index can completely subsume the function f_c of a traditional index. For this, the model learns the sort order or structure of lookup keys and uses these signals to effectively predict the tuples' position or existence. However, this work only allows for good read performance. Data updates can still trigger erroneous predictions. Follow-up works improve upon the concept by optimizing a learned index's write/update performance [45].

Join Enumeration. The second area is join enumeration and is closely aligned to query optimizations. Join enumeration is a long-standing field of research within the database community to solve the complex problem of finding the correct order of joins in a query during or for its execution. Naturally, several works are using ML to improve join ordering. For example, Marcus et al. [55] and Trummer et al. [85] use Reinforcement Learning. This technique learns continuously during runtime to use synchronous capabilities during query execution to be directly used for join ordering. Zhang et al. [102] further improve this concept by adding a supervised learning approach on top of the reinforcement learning, stabilizing the model's results, especially during the first uses of the model where the reinforcement approach lacks quality because it did not see enough data. The supervised approach is learned beforehand and delivers good results right after deployment.

Cardinality Estimation. The third area is also closely related to query optimization: cardinality estimation. Cardinality estimates are essential for nearly all cost-based query planners in DBMS because they support the decision-making of the planner with estimates about the size of filtered base tables or intermediate results. The first work in this area by IBM [53] uses neural networks to learn cardinalities from filter predicates on base tables. This concept was extended by Kipf et al. [44] to incorporate estimates over joins and arbitrary predicate filters using a complex neural network architecture called Multi Set Convolutional Network (MSCN). One requirement for training in both works is a representative user workload. In further research, Hilprecht et al. [36] propose a workload-independent approach for learning cardinalities. They use the predefined key relations in a schema to learn correlated columns and predicates with a specialized model called Relational Sum Product Network (RSPN).

Query Optimizer Hinting. Query optimizer hinting is the fourth area. A *hint* is a knob telling the query compiler which physical or abstract resources it can use during query optimization, i.e., the number of parallel workers or the types of joins. A significant contribution to the field is the Bandit Optimizer (BAO) by Marcus et al. [54]. This optimizer uses reinforcement learning to find the best query plan during runtime. It also learns the costs of different query plans under different hint sets. It uses the DBMS's query planner to generate these alternative plans. The Bandit Optimizer then decides which hint set is the most beneficial for the query, i.e., producing the lowest costs. Hertzschuch et al. [35] learn the best query execution hinting by storing query meta information in tries. In one trie, they store aggregated join and predicate-specific information for a specific set of joined tables. This allows for reusing this meta information for any new query and its optimization according to beneficial hint sets.

Selection Strategies. The last of the five areas in our overview are selection strategies. The task of any selection strategy is to choose one option out of many according to some quality criteria. This work focuses on integer compression selection strategies, especially in column stores, because they benefit the most from a well-rounded decision strategy and can significantly boost query execution [24]. Boissier et al. [11] and Jin et al. [41] both use regression models to directly predict query performance from a given input data distribution of a column. With these workload-dependent approaches, the models can learn to connect the data properties and the impact of compression on query performance. Where Boissier et al. use simple linear regression models, Jin et al. employ complex Long Short Term Memory neural networks. Jiang et al. [40] go one step further by integrating a classification selection strategy into their database CodecDB. The classification is capable of directly predicting the best compression algorithm to be used to obtain the smallest query memory footprint.

2.3 GLOBAL LEARNING STRATEGIES

One can see that there is much research going on in the area of ML support for DBMS, i.e., the replacement of database components with ML models. Of course, this is not a complete listing of all the works or areas, but it gives a general impression of the importance of this topic. In general, all these works share one common characteristic. They use one *global model* to solve the specific problem at hand [93].

Definition. *A **global learning strategy** with a **global model** generates one single function f_m to replace f_c.*

This is depicted in Figure 2.1. There, a single model f_m tries to model the whole problem space P at once. Consequently, any global model is a one-size-fits-all solution. It focuses its model expressiveness to cover all aspects of a problem at once. Hence, we identify four shortcomings.

(S1) The model must be complex to cover the whole problem space P. The more complex the problem, the more neurons, decision trees, or similar elements in a model are needed, making them very slow in training and very large in their memory footprint.

(S2) This complexity leaves the models vulnerable to overfitting. A complex model might not cover the whole problem space P and map the oracle function F completely, especially for unknown data. Therefore, the generalization capability to unknown data, one of the most important properties of an ML model, cannot be assured anymore.

(S3) To compensate for the previous two shortcomings, most approaches use much training data. This leads to extended training and data collection times. For example, generating or collecting example queries for learned cardinality estimators introduces a high load on a DBMS without generating an immediate benefit for the database.

(S4) Global models have slower forward passes through their complexity, which reduces their application speed during runtime. A global model neural network with thousands of neurons and millions of weights needs much calculation to get an output to a given input. Even though these calculations are optimized in most frameworks and are only in the range of two or three-digit milliseconds, time is crucial for all five areas of research mentioned. So, milliseconds for one model application can be too slow, especially if the model needs to make several predictions to get to a solution.

In this work, we want to circumvent these four shortcomings of global complex ML models. For motivation, we look at how a classical DBMS component solves this problem. Our example component is the database index. Indexes are not built globally on the whole schema but on certain single columns or combinations of columns within a table [42]. So, they divide the problem space into smaller sub-problems via *divide-and-conquer*. They lead to better performance in DBMS, which can be attributed to this property [42, 46, 48]. Any index only needs to solve the problem of finding a tuple's position for the columns on which it is built. This is beneficial for the problem space because the position of a qualifying tuple depends on the predicates of a query. Predicates access columns and change the set of qualifying tuples in a table. Therefore, using the columns to divide the problem space is evident. It helps to lessen the problem's and the index's complexity generating better performance in this particular use case. The lesser complexities are the core aspect that helps to improve query performance with indexes.

Therefore, we need to define a general set of goals or advantages for learned approaches to reduce the complexity of the ML models replacing the DBMS component. We can derive them directly from the four shortcomings of global models and the motivational index example. Any learned approach for DBMS components should have these four advantages addressing the shortcomings of global models:

(A1) The approach's models should be small in complexity and, therefore, fast in training and small in memory footprint.

(A2) The alternative models should be less prone to overfitting by concentrating their full expressiveness on smaller portions of the problem space P.

(A3) Non-global models should require less training data. The smaller amount of training data and the lesser complexity lead to faster training times than global models.

(A4) Non-global models should be very fast in their application. Simple model structures would allow for very fast forward passes that only add little overhead during runtime.

In the remainder of this work, we will address the four shortcomings of global models and the four advantages of alternatives as argumentation for any presented concept or application. We will derive an alternative to the global learning strategy, called a local learning strategy, from an example DBMS component, which we replace with learned strategies. The derivation of the local learning strategy will follow the argumentation of the shortcomings and advantages to an extent where it outperforms the global strategy in the experimental evaluations.

2.4 SUMMARY

In this chapter, we presented the DBMS as a complex adaptive system and the possibilities to replace database components with learned equivalents. There exists a plethora of related work in this area. However, most of these approaches model the problem globally and share the same characteristics and, therefore, limitations. We derived suitable prerequisites for an alternative approach without the global models' limitations. In Chapter 3, we motivate and present the alternative local approach and how it is able to provide the advantages **(A1)** to **(A4)** by using it for ML-based cardinality estimation. Furthermore, the alternative strategy is applied to query compiler hinting and selection strategies for integer compression in Chapters 4 and 5. These chapters show that this approach does not only work for one use case but can be seen as a general approach for solving problems with ML.

3

THE LOCAL LEARNING STRATEGY FOR CARDINALITY ESTIMATION

Query optimization is still a significant challenge due to ever-increasing data sizes, whereby nearly all query optimization techniques are cost-based [13, 57]. This makes cost-based optimizers the most important type of optimizers. In cost-based approaches, cardinality estimation plays a dominant role by approximating the number of returned tuples for every query operator within a query execution plan [13, 57, 62]. These estimations are used within different optimization techniques for various decisions, such as determining the best join order [30], choosing the optimal operator variant, or finding the optimal placement within a heterogeneous hardware environment [74]. For this reason, it is essential to have cardinality estimations with high accuracy.

Unfortunately, most traditional estimation approaches, which are based on statistical models with strong assumptions, are not accurate enough because of these assumptions [49]. Here, the main critical assumptions are uniformity and data independence [57, 79]. However, these estimators are still commonly used in today's DBMS. For example, the color *red* is usually uniformly distributed over all car brands, but its distribution for the manufacturer *Ferrari* is somewhat skewed since most of them are red. In this case, the color and the manufacturer are highly correlated in an non-uniform way leading to erroneous cardinality estimates using traditional approaches. Another example would be the correlation between students and their age. Most undergraduate students are 24 years old or younger. So, the columns *age* and *status* in a university members database are highly correlated. A query asking for all students with an age smaller than 25 over the two tables `R(id,A,B)` and `T(id,C,D)` with columns `R.A` for age and `T.D` for status will help throughout this chapter to explain different cardinality estimator and their performance under correlated data. Figure 3.1 gives an overview over the whole database schema including all foreign key relations between the six tables. The example query can be described with:

```
SELECT *
FROM R,T
WHERE R.id = T.id AND R.A < 25 AND T.D = 'student';
```

The `id` column contains the joined key relation. The columns `R.B` and `T.C` contain different data and are of no concern for this example. The query result set cardinality is ten. The database schema contains the additional tables `S`, `U`, `V`, and `W` that are also out-of-scope for this query.

3.1 PRELIMINARIES FOR LEARNED CARDINALITY ESTIMATION

Over the decades, the database research community has brought forward many cardinality estimation techniques. For example, many synopses-driven approaches were integrated, from sketches [31, 77] to histograms [60, 81]. Sampling techniques were introduced for cardinality estimation as a versatile alternative, but selective predicates cause

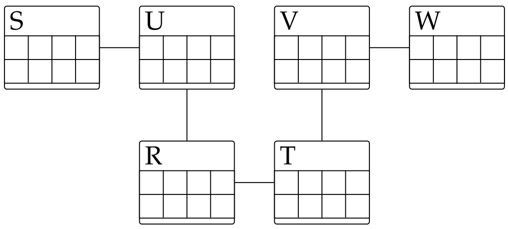


Figure 3.1: Schema for the example query.

inaccurate estimates [20, 88]. A promising way to overcome the limitations of these previous or traditional works is the use of machine learning, including neural networks, for cardinality estimation [36, 44, 53, 55, 69]. Here, the central assumption is that a sufficiently complex ML model can generalize complex data dependencies and correlations. In general, ML is *arbitrary function approximation*. The function that underlays the *cardinality estimation* problem in databases is

$$\textbf{query} \times \textbf{data} \rightarrow \textbf{cardinality} \tag{3.1}$$

To incorporate the limited input capabilities of ML models, we approximate Equation 3.1 by the two-step mapping

$$\textbf{query} \times \textbf{data} \ \rightarrow \ \textbf{vector} \ \rightarrow \ \textbf{cardinality} \tag{3.2}$$

where the vector is a numerical representation of any SQL query and the underlying data. ML models as arbitrary functions mapping feature vectors X to a *label* or *target* y can be formulated as:

$$f\colon X \rightarrow y, X \in \mathbb{R}^d, y \in \mathbb{R} \tag{3.3}$$

The number of features d is the *dimension* of the feature space. This makes cardinality estimation a *regression problem* because it predicts a continuous variable, i.e., the cardinality. Therefore, the process for any supervised ML-based solution for cardinality estimation is the same. It follows the general concept with training phase and forward pass as described in the Introduction. Figure 3.2 shows the general steps to design models capable of replacing DBMS components, i.e., estimating cardinalities. After selecting a workload of queries, this workload is executed against the DBMS to collect the true cardinality. This is called *labeling*. At the same time, the query is mapped to a numerical vector in a step called *featurization*. This step is required because ML models have limited input capabilities and only work on numeric input. By comparing the true cardinality with the predicted cardinality for several hundred or thousand queries, the model learns complex data dependencies and correlations. In this *training* step, the model is changed according to the errors it produces on the training data. For the training, it is important that the mapping from feature vectors to labels or targets is deterministic and that the same input produces the same target all the time:

$$\forall x_1, x_2 \in X\colon x_1 = x_2 \Rightarrow f(x_1) = f(x_2) \tag{3.4}$$

It is catastrophic for an ML model if the same input leads to different labels. This leads to a low model quality either by an averaged output or by outputting a specific label for all

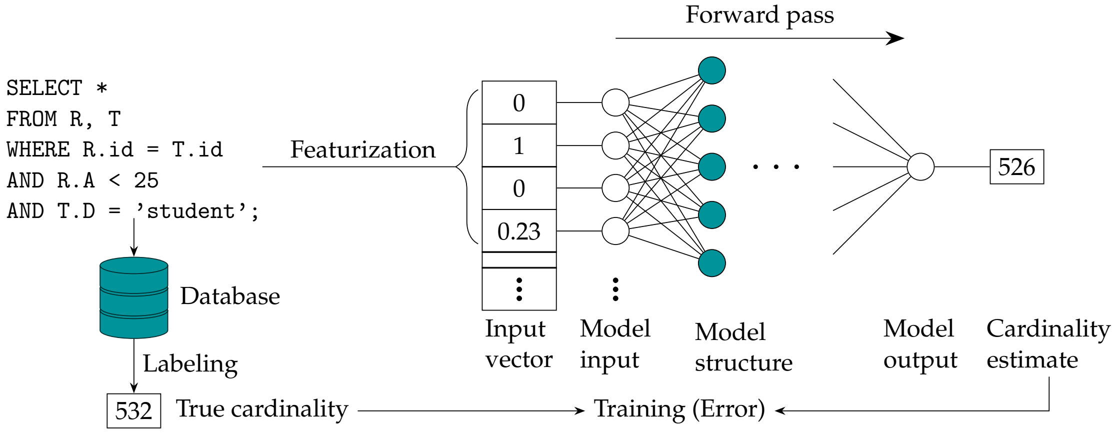


Figure 3.2: Process for cardinality estimation with learned models.

inputs. Here, the entropy is maximal and the model cannot learn any information. Therefore, query featurization must be collision-free to avoid these problems. Once training is finished, the model can be deployed and used for new unknown queries. This *forward pass* is orders of magnitude faster than the training and is therefore the true advantage of supervised ML models for complex tasks. It relays the complexity of the problem from the decision making during runtime to the separate training before the application of the model. But not every model is equally fitting for every task. The models' quality highly depends on the featurization and the models' complexity. A rule of thumb is: The more complex the problem, the more complex the model needs to be. Therefore, many models used in cardinality estimation are getting more and more complex whilst their computational complexity also increases. This leads to longer training times and forward passes and influences query execution performance in a negative way. We call this tendency towards large specialized ML models for cardinality estimation a *global model* trend [93].

In the following Section 3.2, we will show how traditional approaches and global ML models are used for cardinality estimation and what are the most common four shortcomings of these works. Next, we show how a new local learning strategy is used for improved cardinality estimation circumventing these shortcomings in Section 3.3. Section 3.4 details the necessary prerequisites and examples of ML models to be used as local models. We demonstrate the importance of query featurization in Section 3.5. In Section 3.6, an excursion into query labeling shows a performance optimization for the most compute-expensive part of our approach. Section 3.7 shows an extensive evaluation of our local learning strategy. Lastly, a summary gathers all advantages of local models for cardinality estimation in Section 3.8. For the whole chapter, we use the research results from [63, 90, 91, 93, 94].

3.2 RELATED WORK

This section gives an overview over different cardinality estimation techniques. We start wit a traditional estimator before moving on to learned solutions. There, we present two different approaches: one based on a neural network and one based on a sum-product network. Throughout the section, we will show the advantages and disadvantages of all approaches. Lastly, the disadvantages are collected to form four general shortcomings of the learned solutions.

3.2.1 System R

The first cardinality estimation technique *System R* was established by Selinger et al. [79] and is commonly referred to as the *independence assumption*. For instance, the DBMS *PostgreSQL* implements this estimator. This technique is based on per-selection-predicate estimates that rely on uniformity assumptions, i.e., it is assumed that each value in the domain of a column has the same frequency. The overall estimate is calculated as the product of the per-selection-predicate estimates. For joins, a selectivity that is based on referential integrity and a key/foreign-key join relationship is computed and multiplied with the overall selectivity estimate. It is well known that these assumptions are often violated in real-world data sets since the frequency distribution in many attributes is skewed and many pair-wise attributes are highly correlated [49]. The limitations of the assumption of independence for correlated data can also been shown for our example query. Let $c(R_\sigma)$ be the cardinality of a relation after applying a selection filter σ and $c(R)$ the total number of tuples in `R`. The selectivity $s(R_\sigma)$ of a filter on `R` is defined as:

$$s(R_\sigma) = \frac{c(R_\sigma)}{c(R)} \tag{3.5}$$

For our example, we assume selectivities of $s(R_{\sigma:A<25}) = 0.1$ and $s(T_{\sigma:D=\text{student}}) = 0.05$. The relations contain $c(R) = c(T) = 100$ tuples with $d(R) = d(T) = 100$ distinct entries in both tables. The cardinality for the join $c(R \bowtie T)$ can be derived as:

$$\begin{aligned} c(R \bowtie T) &= c(R) \cdot c(T) \cdot \frac{s(R_{\sigma:A<25}) \cdot s(T_{\sigma:D=\text{student}})}{\max(s(R_{\sigma:A<25}) \cdot d(R), s(T_{\sigma:D=\text{student}}) \cdot d(T))} \\ &= 100 \cdot 100 \cdot \frac{0.1 \cdot 0.05}{\max(0.1 \cdot 100, 0.05 \cdot 100)} \\ &= 5 \end{aligned} \tag{3.6}$$

The multiplication of the two selectivities for `R` and `T` is the direct implementation of the assumption of independence over the two columns A and D. However, as we know, the status of person as student highly correlates with the age of that person. As mentioned before, the true cardinality of the join $c(R \bowtie T)$ is ten because all ten students from table `T` are probably younger than 25. Additionally, the multiplication of the selectivities with the number of distinct values is called the *uniformity assumption* where the values in all columns are considered in an uniform distribution before and after filtering. This example shows that for correlated and non-uniform data, System R can produce misestimates orders of magnitude smaller than the true cardinalities. Many DBMS use frequency histograms to avoid the uniformity assumption [39, 67]. However, multiple estimates are still multiplied, based on the independence assumption. Multi-dimensional histograms capturing the correlations and distributions over several columns at once exist but are uncommon because they are expensive to compute [14].

3.2.2 Learned Estimators with Neural Networks

To solve the issues of System R in a more sophisticated way, there have been works on formulating cardinality estimation as a supervised machine learning problem [44, 53]. On the one hand, the authors in [53] proposed learned cardinality estimators for single tables. They train a simple neural network to predict a base table by directly encoding every predicate directly via its predicate filter value. However this approach is limited to single or base tables only and does not cover joins. On the other hand, Kipf et al. [44] introduced a specialized model architecture called multi-set convolutional neural network (MSCN) which models cardinality estimation with one global model. The neural network processes three inputs derived from an SQL query independently. These are the joined tables (*table set*), the join keys (*join set*), and the chosen predicates on the used tables (*predicate set*). Additionally, MSCN uses samples of the first 1,000 rows under the queries filter predicates as a bitmap giving the truth values of the query's predicates. MSCN is capable of modeling joins and predicates over several tables and hence can cover correlations in the data. The complex network structure estimates the cardinality of the given query as shown in Figure 3.3. The three input sets are folded upon themselves in several layers of convolution. After averaging and concatenating the output of all three convolution components, fully-connected layers predict the cardinality of a query. The complexity of the MSCN is mainly defined on the number of layers in the convolution and fully-connected components. In the original work, this is to set to four layers per component with 256 neurons each.

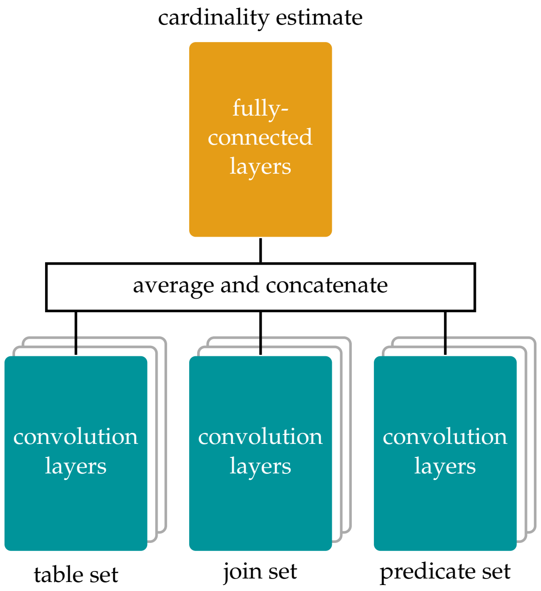


Figure 3.3: The multi-set convolutional neural network (MSCN) architecture.

We use our example query from the beginning of this chapter to illustrate the workings of the MSCN. The general idea is the *featurization* of our query into the three sets.

$$\text{table set: } \{[0\,0\,1\,0\,0\,0 \,|\, \mathit{samples}], [0\,0\,0\,1\,0\,0 \,|\, \mathit{samples}]\} \tag{3.7}$$

$$\text{join set: } \{[0\,0\,1\,0\,0]\} \tag{3.8}$$

$$\text{predicate set: } \{\underbrace{[\overbrace{1\,0\,0}^{\text{op id}}\ \overbrace{0.3}^{\tilde{x}}]}_{\texttt{R.A}}, \underbrace{[0\,1\,0\,0.6]}_{\texttt{T.D}}\} \tag{3.9}$$

The table set comprises the one-hot-encoded enumeration of joined tables. From Figure 3.1, we see that R and T are the third and forth table out of a total of six tables in the schema. Therefore, a bit vector for each table is created where the n-th bit is set to one marking that the n-th table was used in this query. Additionally, *samples* represents the first 1,000 qualifying rows as a bit vector of the same length for R where `R.A < 25` is true. The same goes for T with `T.D = 'student'`. The vectors for R and T are put into an enclosing set and define the first input to the MSCN. Analogously, the join set encapsulates all joins as one-hot bit vectors. The n-th join is represented by the n-th key relation as the n-th bit in a single vector. For our example query, the join R ⋈ T is on the third out of five key relations in our example schema. Again, this is visualized in Figure 3.1. The predicate set collects all predicates used in the query. However, it decouples the filter from the table it is applied to. This connection is already modeled by the *samples* in the table set. First of all, the predicate set ignores any column that has no filter on it. In our example, these are the columns `R.B` and `T.C`. For any filter on the columns `R.A` and `T.D`, a vector is generated in the following way. From the operator table (cf. Table 3.1), choose the fitting bit encoding for the operator in the filter. Filters with an operator other than the listed ones are ignored.

Neural networks tend to work better with input values between zero and one [76]. To map our neural network inputs to this range, we calculate the normalized value $\tilde{x}$ for the filter predicate value x over the column c.

$$\tilde{x} = \frac{x - \min(c)}{\max(c) - \min(c)} \tag{3.10}$$

Table 3.1: Operator table

Operator	Op id
<	100
=	010
>	001

Non-numerical columns, like strings, are dictionary-encoded and then normalized. Operator ids and normalized values are concatenated into one vector and put into the predicate set. After generating these three sets a learned MSCN passes them through its architecture and predicts a cardinality for our query. Of course, for this to work, many featurized and labeled queries are required during training. However, the inner workings of the prediction are not traceable through the opaque nature of the underlying neural network.

This directly leads to some drawbacks of the MSCN. The large problem space to cover the whole schema leads to a sparse featurization of different queries. Furthermore, if an unknown query without a representative during learning is passed to the network, the network's interpolation capability fails and the estimate is erroneous. We will detail this general problem of all global models in Section 3.3. The complex network structure also results in an increased learning time of the model and slower cardinality estimation itself. Additionally, retrieving the qualifying samples for the table set are expensive to obtain.

3.2.3 Learned Estimators with Sum-Product Networks

The work by Hilprecht et al. [36] wants to decouple the dependence of a learned cardinality estimator on a (user-)workload. They argue that while data can be obtained easily, the retrieval of workloads from clients or users is much harder because of data privacy and intellectual property concerns. To reach a workload-independent concept, the authors propose a specialized model called the relational sum-product network (RSPN), which expands upon traditional sum-product networks (SPN). A SPN can be seen as the successor to neural networks in spirit. It is a tractable representation of probability distributions compared to neural networks that do not have probabilistic semantics [71]. The representation is a tree containing sum, product, and input nodes according to the equation for the joint distribution modeling a prediction of a value or class [72]. Through their structure, SPNs partition the distribution of the data into sub-distributions and can derive a prediction from the forward pass. Learning is done through independence test between the data partitions. For the relational part of the RSPN, the data tables in a database are partitioned into single columns and their data distributions. For our example tables `R` and `T` from the schema in Figure 3.1, a corresponding RSPN for columns A to D and their histogram distributions is depicted in Figure 3.4. It shows different distributions for `R.A` and the correlated distributions for `T.D`. This is just a part of the whole RSPN of the schema, which would contain all columns and correlated distributions.

The weights along the edges are learned during training by joining tables along their key relations and retrieving the cross-correlations under different filter predicates between all columns. The network learns the probabilities between all column histograms from the join results. This is done by recursively splitting the data in different row clusters (sum node) or clusters of independent columns (product node). This is similar to the construction of multidimensional histograms, which is very compute intense. Therefore,

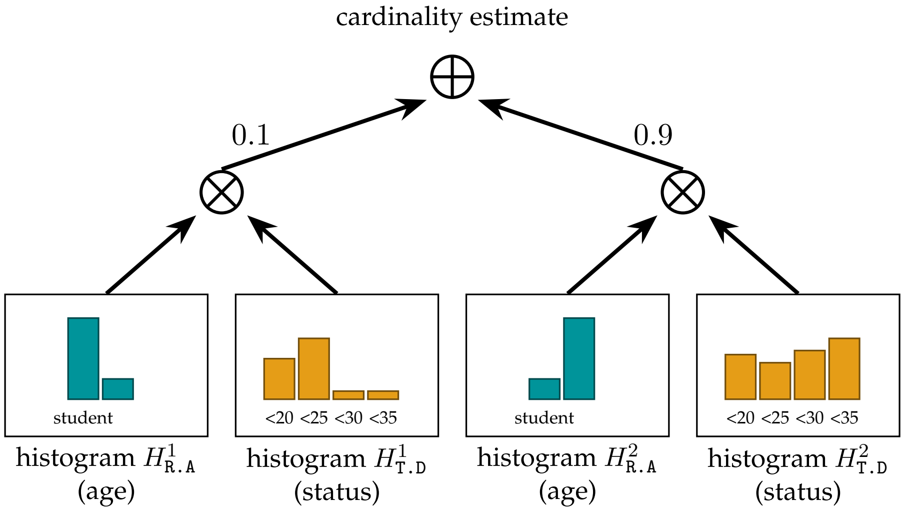

cardinality estimate
0.1
0.9
student
<20 <25 <30 <35
student
<20 <25 <30 <35
histogram $H^1_{\texttt{R.A}}$ (age)
histogram $H^1_{\texttt{T.D}}$ (status)
histogram $H^2_{\texttt{R.A}}$ (age)
histogram $H^2_{\texttt{T.D}}$ (status)

Figure 3.4: The relational sum-product network (RSPN) for columns `R.A` and `T.D`.

it is recommended to only use table samples for the joins to limit complexity and computation time. However, the final RSPN uses less memory than a comparable multidimensional histogram in all cases. During application, the filter of every predicate is applied to the respective histograms in the RSPN. The resulting selectivities are multiplied $\bigotimes$ and added $\bigoplus$ through the network. Finally, they are multiplied by the total number of rows in the full join of all tables over their key relations. For our example query the cardinality would be resolved like

$$\begin{aligned}c(R \bowtie T) =&(0.1 \cdot s(H^1_{\texttt{R.A = student}}) \cdot s(H^1_{\texttt{T.D < 25}}) \\ &+ 0.9 \cdot s(H^2_{\texttt{R.A = student}}) \cdot s(H^2_{\texttt{T.D < 25}})) \cdot c(R \bowtie_{id} T) \\ =&(0.1 \cdot 0.8 \cdot 0.7 + 0.9 \cdot 0.1 \cdot 0.4) \cdot 100 \approx 9\end{aligned} \tag{3.11}$$

Thus, a SPN is tractable compared to a MSCN or any neural network. This is an advantage because SPN prediction are inherently explainable. However, the problem of the complex and time-consuming training process through partitioning the all joined tables, including their joined results, in a schema remains. Sampling reduces this overhead, but might reduce the quality of an RSPN by omitting information about the underlying data distributions.

3.2.4 Shortcomings of Global Strategies

Besides the individual drawbacks of all approaches, there is one common disadvantage. Most available techniques are based on a *global approach* by creating a single complex model over the entire problem space, i.e., the database schema. MSCN and RSPN construct one model to cover all joins and tables at once. This concept is detailed in Figure 3.5 over our example schema from Figure 3.1, where tables `R`, `S`, `T`, `U`, `V`, and `W` are the complete schema in a database. The global model spans the whole schema. This leads to four disadvantages **(S1)** to **(S4)**.

(S1) Large complex models: The resulting global model for cardinality estimation can become large in structure to capture as many aspects of the schema as possible. Figures 3.3 and 3.4 detail this structural complexity. MSCNs can have millions of neurons and weights reaching several megabyte in memory [63]. From a memory footprint perspective, it is desirable to reduce the consumed memory of an ML model component because

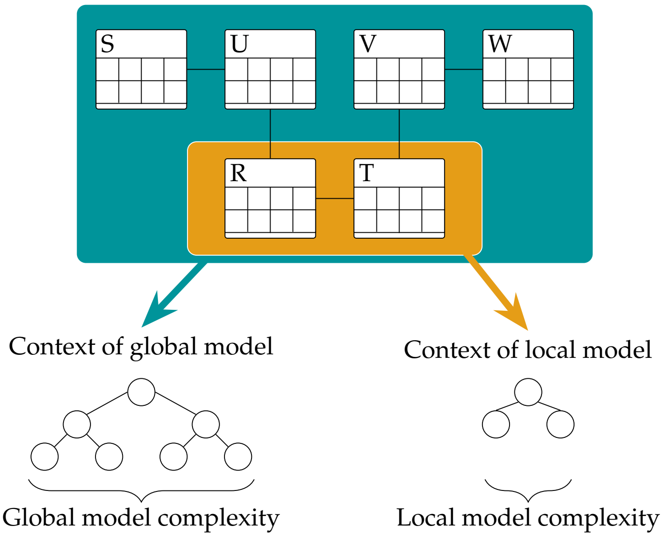


Figure 3.5: Global and local approach contexts and their model complexities.

DBMS components should be very memory efficient. For example a simple index over gigabytes of data usually only requires kilobytes in memory itself [42].

(S2) Overfitting: The global model is trained on sampled queries from the entire schema. However, this leads to sparse problem representations because the problem space cannot be covered completely due to the fact that one cannot sample all necessary queries in the problem space. Building upon Figure 2.1, Figure 3.6 depicts this problem by adding points as representatives for sampled example queries for training. The problem space for the global model on the left is not covered with example queries in an even way. Therefore, the global model struggles to generalize over the whole problem space. A request for the model to estimate an unknown query outside the function's coverage f_g will lead to erroneous estimates, which we wanted to avoid in the first place.

(S3) Amount of training data: A measure to counteract the previous point is to sample more queries over a wider area in the problem space. However, the number of possible training queries becomes extremely large, even for a limited number of tables and predicates. For our example, the schema with six tables with three columns, each containing 100 possible values and three possible predicate operators ($<$, $=$, $>$) would generate a problem space of $(2^6 - 1) \cdot 2^3 \cdot 100 \cdot 3 = 151{,}200$ queries, which is a lot given the small problem at hand. Sampling every query for training would be quite expensive because every query needs to be run against the database to retrieve its true cardinality, as introduced in Figure 3.2.

(S4) Slow forward passes: Hand in hand with the complex structure comes the negative effect on the forward passes in a global model. Every cardinality estimate requires a complete pass to the whole model. The more complex the model, the longer this pass takes. Through its complexity, the MSCN takes 33ms for a single request to the model to estimate one query's cardinality [93]. This is an important overhead to be considered because the forward pass is done during query execution and has a direct influence on the query runtime.

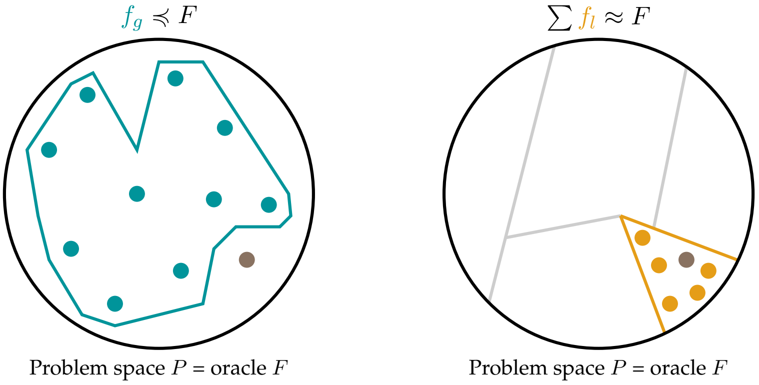

$f_g \preccurlyeq F$
$\sum f_l \approx F$
Problem space P = oracle F
Problem space P = oracle F

Figure 3.6: Coverage of problem spaces by global f_g and local models f_l.

3.3 THE LOCAL LEARNING STRATEGY FOR CARDINALITY ESTIMATION

As promised in the problem description, we propose a cardinality estimation technique focusing on model structures avoiding the four disadvantages of global learning strategies by using divide-and-conquer. Like any other divide-and-conquer solution, the general concept is splitting the problem space into sub-problems and solving each sub-problem with less complex models. Therefore, unlike global learning strategies, we do not learn one global model but a distinct model for each well-defined sub-problem. The practical motivation behind our approach is the motivational index example from Section 2.3. There, we motivated the use of divide-and-conquer by the example of database indexes, which focus on certain columns to retrieve qualifying tuple positions. To adapt the indexes' behavior to cardinality estimation, we have to find an underlying split criterion, like columns for indexes, for this problem. For cardinality estimation of queries, each set of joined tables represents a self-contained sub-problem since queries on these tables are reasonably homogeneous with respect to the joins and differ only in the filter predicates. For each such self-contained sub-problem, we learn a separate model for fine-grained problem distinctions, which would otherwise be overshadowed by queries of completely different sub-problems. An example of a sub-problem is any sub-part of a schema, i.e., a single join, as depicted in Figure 3.5.

Definition. *A* ***local learning strategy*** *deploys* ***local models*** *focusing on smaller sub-problems of the whole problem space P.*

Each local model is always specialized to a specific sub-problem and can generalize within this specific smaller sub-problem space. The local models cover the same problem space in sum but with less complexity per model and, therefore, better quality and performance. These two improvements stem from the general divide-and-conquer characteristics of our approach [10, 46, 76].

So, the crucial point of the *local learning strategy* for cardinality estimation is dividing the problem space into sub-problems, also called *contextualization*. This is equivalent to the *placement* of the local models. To reach the maximum effectiveness of our approach, the placement of local models needs to fulfill two criteria: (1) the collection of all models should span the whole problem space and (2) no sub-problem should completely subsume another sub-problem.

Definition. *A sub-problem of a problem space P according to divide-and-conquer containing a single local model is called a* ***context****.*

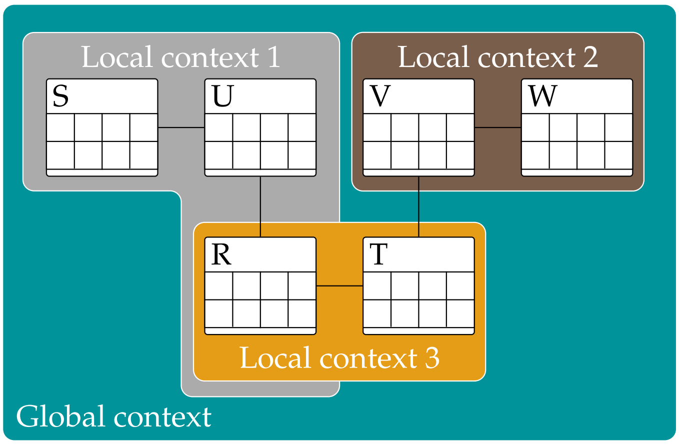


Figure 3.7: Global and local approach contexts and their placement.

Similar to indexes, cardinality estimation depends on qualifying tuples defined by a query on a database schema. Therefore, we need a partitioning attribute for query workloads accessing a database schema. Similarity attributes of analytical queries are their common elements: selected columns, projected columns, and joined tables. Selected and projected columns have many properties that directly influence the properties of a subproblem partition itself, like filter predicate information. So, using them makes the partitioning ambiguous. However, the diversity of query structures in a workload is usually limited since these queries work on the same relational database schema with a finite set of tables. Thus, workload queries can be partitioned into groups based on the joined tables they access. This is a stable proposition because the set of joined tables does not influence intra-query properties compared to filter predicates on selected columns. Even more specifically, the queries only differ in their filter predicates within such a partition. Therefore, in cardinality estimation, a single *context* contains all schema-specific workload queries with the same set of joined tables. In the example from Figure 3.5, the single local model spans all queries accessing the context defined by the join of tables `R` and `T`. However, the schema sub-part for a local model can be any number and combination of joined tables. Without loss of generality, we focus on equi-joins. To span the whole problem space according to this definition, there is the possibility that the contexts overlap or include each other, but two contexts never contain the same exact set of joined tables. Therefore, the two criteria from the beginning are ensured. The local models span (1) the whole problem space and (2) do not subsume each other. Because the set of accessed tables is fixed for a specific context, a local model can spend more expressiveness to solve the complexity of different predicates. A local model does not need to model the join properties explicitly but can model them implicitly through correlations between predicates for a specific context and its tables. However, we would like to point out that the rules for constructing contexts have to be adapted for each problem individually. While the joined tables are important for cardinality estimation, indexes, in contrast, focus on columns and their combinations. A context creation for a learned index would have to include these properties accordingly, but this is out-of-scope for this work.

To further deepen the presentation of contexts, Figure 3.7 shows our example schema divided into three contexts according to our context definition. The current placement covers different sets of joined tables according to their key relations. The actual context instances, i.e., which tables should be handled by a single model, must be derived from the user workload. In our example, the workload contains queries that join `R`, `S` and `U`; `V` and `W`; and `R` and `T`. This leads to an overlap between the first and third context. Contrarily, there is no context for `T` joining `V` because the example user workload did not contain any queries with this particular join. The *user workload-oriented* creation of contexts fulfills both criteria for local model placement. It does span the whole problem space of the user workload and no context is completely included in another context. The last criterion is met by checking if the joined tables of a query are a subset of a set of joined tables of an existing context. Otherwise, a new context is created.

Given this straightforward heuristic for placing models, we examined other possible placement strategies [90]. There, we detail that the metric that improves the placement of models in context is a combined modeling with histograms and local models. We use the correlation between all columns in a context to decide where to instantiate a learned model over a histogram. This is directly derived from the shortcomings of traditional histogram-based estimators, whose quality deteriorates if highly correlated data is used. However, histograms still work on uncorrelated data, can be used in the same contextualized way as local models, and are faster in setup than learned models. Even though this combined approach improves the overall quality of the estimates, retrieving the correlation over the superset of all columns in a context takes too long. Therefore, we argue that this correlation-based heuristic is only feasible if there are no setup time limitations. Additionally, if a heuristic-based placement is already slow, a learned placement on top of the local learning strategy would not suffice in real-world applications because creating contexts would take even longer.

Advantages of the Local Learning Strategy

In this section, we presented the divide-and-conquer design of a *local learning strategy* for cardinality estimation that overcomes the four shortcomings **(S1)** to **(S4)** of global models. For this, it offers the following four advantages **(A1)** to **(A4)** directly corresponding to the four shortcomings.

(A1) Less complex models: Each local model solves a sub-problem of the global problem space. Therefore, they need less expressiveness to reach the same quality in estimates as the global model in the same sub-problem space. Building a monolithic global model underrepresents parts of certain estimates in the training data because a large global model cannot include influences that occur seldom or outside its function space f_g. This leads to poor performance for estimates of queries outside this function space. With the partitioning of the whole problem space, we can use more of the local model's expressiveness to estimate cardinalities special to this sub-problem. Of course, the partitioning needs to be done in a way that builds orthogonal sub-problems with similar characteristics according to the partitioning attributes.

(A2) Better quality: Another advantage of our local approach is that the problem representation gets less sparse. Given our example model from Figure 3.5 concentrating on the context with the join between R and T, we get a sub-problem space of $(2^2-1) \cdot 2^3 \cdot 100 \cdot 3 =$ 7,200 queries, which is a reduction by 95% in problem space complexity compared to the $151{,}200$ global model queries. The higher coverage allows a learned model to generalize its prediction, producing higher quality in estimates. Thus, the coverage of a sample would increase because the same amount of sampled queries would be less sparse for the local sample space. The right side of Figure 3.6 shows this argument for a single local model. If the same unknown query is requested as for the global model, a simple local model covering the corresponding sub-part is now able to estimate the cardinality. Of course, several small local models need to be trained to cover the whole problem space, but this is easier to reach than with a global model as we will show in the evaluation.

(A3) Faster training: Through the lesser complexity of local models, we need less training data. If we sample 1,000 queries in both sample spaces, we cover ca. 0.7% of the global sample space but ca. 14% of the local sample space. Sampling 1,000 queries needs the same amount of time in both scenarios, but the coverage of seen queries with different cardinalities is higher for our local approach. The other side is that we can reduce the number of training queries to reach the same or even better qualities in the local model's sub-part of the schema than the global model. We do not need to sample the full sub-problem space anymore, because its complexity is reduced to a level where less training data is enough to build expressive models.

(A4) Faster forward passes: Alongside smaller model complexity, there comes faster forward passes. The passage of a feature vector through a model, e.g., a neural network, is faster because fewer calculations are required to produce a prediction. Our local models are only a fraction of the size of a global model and their application times during runtime are smaller adding fewer overhead during query processing.

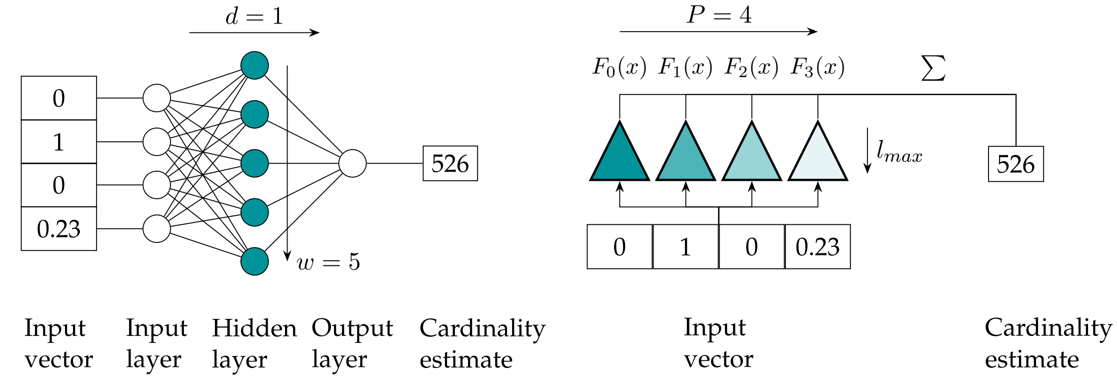
The division of the schema into sub-parts via the contexts, divides the problem space of cardinality estimation over the whole schema to less complex sub-problems. The complexity within a context only depends on the used filters in queries. These filter predicates can be modeled easier in the numerical representation, i.e., featurization, and reduce the complexity of the estimation problem for a single context model. The complexity of different joins is implicitly modeled by the contextualization because a context limits the possible joins to the set of tables defining the context. So, the modeling capabilities and qualities for a context depends on three things, as depicted in Figure 3.2: (1) the local model structure, (2) the featurization of a query, and (3) the efficient labeling of queries with the true cardinalities. In the following, we will go through these three properties and explain how they influence the quality and performance of a local learning strategy for cardinality estimation.

3.4 LOCAL MODEL STRUCTURE

Up until now, we talked about the flexibility of the local model strategy regarding placement. However, through the dynamic placement and the required simplicity of models in each context, we are able to use any standard architecture model. In contrast, MSCN [44] is a highly specialized architecture as presented in Section 3.2. Its expressiveness comes solely from the fact that it models the whole schema in its very complex internal structure leading to the common disadvantages (S1) to (S4) of global models. For our strategy, we want to be model independent, meaning we can use any standard model as context models. The same follows for the query featurization. Going forward, we present two types of ML models that are standard and commonly used for ML problems in DBMS [27, 47, 93]. We will use these to show that we can use any model and featurization combination and produce better estimates than related work in Section 3.7.

Neural Networks

Neural networks (NNs) are able to solve supervised ML problems by modeling the arbitrary function mapping within the problem as a black box [76]. With increasing hardware performance over the last years, their training has become feasible on a large scale. Therefore, it seems to be evident that the database community relies on neural networks to solve the complex problem of cardinality estimation. Here, we focus on multi-layer perceptron or feed-forward networks because they are the simplest form of neural networks. These specific NN are defined by a an input layer, an output layer, a depth d, and a width w. The depth describes the number of hidden layers in the NN, and the width defines the number of neurons per hidden layer. Both variables are critical *hyperparameters* impacting the quality of the model. If all neurons of any layer are connected to all neurons in the next layer, we call such an NN *fully connected*. An NN is depicted in Figure 3.8a as an example model predicting cardinalities.



(a) A feed-forward neural network architecture. (b) A gradient boosting model architecture.

Figure 3.8: Local model structures used for cardinality estimation.

Gradient Boosting

It has been observed that neural networks can be too complex and their training time takes too long. Hence, [27] proposes to use smaller and faster models, based on Gradient Boosting (GB). GB is an ensemble model where weak learners are learned based on the residuals of a preceding learner [58]. For predictions, the GB estimator sums over P weak predictors F_p, each with weight λ_p, and adds a constant c.

$$\hat{f}(x) = \sum_{p=1}^{P} \lambda_p F_p(x) + c \tag{3.12}$$

In our case, each weak predictor F_p is a simple decision tree. The structure of a GB model is also defined by hyperparameters. Here, these are the number of weak learners P and the max number of leaves per decision tree l_{max}. The tree-based structure of our GB models makes them very fast in training and forward passes. This leads to improved estimation accuracy and faster setup times per model.

Both NN and GB are can work with any vector representation, i.e., for a fixed input vector length, they can work with any numeric vector presented to them as depicted in Figure 3.8. This allows us to vary the featurization without having to modify the models' architecture. Here, this is an advantage over complexly structured estimate models, like MSCN, because it is straightforward to modify NN and GB to use any arbitrary input featurization. Finally, we also could test simpler models, like *linear regression* and *support vector regression*. However, we do not include these ML models in the further discussion and evaluation since their estimates are assumed to be worse by a significant factor.

3.5 QUERY FEATURIZATION

To get the most out of the flexible input capabilities of both of our model structures, we employ different query featurization to model different aspects of a query in the input vectors. *Query featurization* is the process of representing a query in a numerical vector, also named *feature vector*. This is important because ML models are only capable of using numerical input. The quality of a featurization can be measured by its impact on the model quality when using the featurization. The better a feature vector represents the predicates in a query, the better the model quality, the better the query featurization. A query featurization is of high quality if the back-transformation from a feature vector to a query generates the same original query from which the feature vector was derived. In other words, a query featurization is lossless if the feature vector is as expressive as the query. The lossless property is not either satisfied or violated by some featurization in general. Instead, a query featurization satisfies the lossless property for a certain class of queries. When the lossless property for a class of queries is not met, information loss occurs, since an ML algorithm, to which feature vectors serve as input, cannot distinguish between different input queries with the same feature vector representation. Therefore, query featurization needs to include the important expressive parts of a query to be useful for an ML approach. For cardinality estimation with local models, the queries need to be featurized in a way that includes filter predicate information because filter predicates have an high impact on the cardinalities of queries. The join order, also having a high impact on cardinalities, is modeled implicitly via the context and does not need to be included in the featurization. This section presents four query featurization techniques with different central points and expressiveness as presented in [63]. The scope for all four featurizations is restricted to predicates, where an attribute is compared to a filter value using one of the operators $\{=, >, <, >=, <=, !=\}$. We consider conjunctions as well as certain disjunctions of predicates. We also consider arbitrarily many predicates per attribute.

Singular Predicate Encoding

Singular Predicate Encoding (SPE) was first described by Kipf et al. [44]. We extended the featurization in [93]. As the name suggest, SPE uses single occurrences of predicates and encodes them numerically. To encode a predicate like `A < 25` from our example query, the predicate is split and encoded into three parts. Suppose our table `R` has the following data ranges: $\min(\texttt{id}) = 1$, $\max(\texttt{id}) = 50$, $\min(\texttt{A}) = 0$, $\max(\texttt{A}) = 115$, and $\min(\texttt{B}) = 0$, $\max(\texttt{B}) = 12$. For simplicity, all attributes have data type integer.

(1) Attribute `A` is position-encoded in a one-hot vector, like 010, because `A` is the second attribute column in table `R`.

(2) The predicate value 25 is min-max-normalized according to Equation (3.10) to $\frac{value - \min(\texttt{A})}{\max(\texttt{A}) - \min(\texttt{A})} = \frac{25-0}{115-0}$, which is always a float in $[0, 1]$.

(3) The comparison operator < is encoded to a 3-entries binary vector 001, where each entry represents one of $\{=, >, <, >=, <=, !=\}$ like in Table 3.2, which is an extended version of Table 3.1.

The featurization of `A < 25` could then look like this:

$$\underbrace{010}_{\texttt{A}}\underbrace{001}_{<}\underbrace{0.22}_{25}$$

Table 3.2: Extended operator table

Operator	Op id	Operator	Op id
<	100	<=	110
=	010	>=	011
>	001	!=/<>	101

While [44] average the representations of per-predicate featurizations, we modified the approach with fixed vector lengths [93]. Thus, we can drop the one-hot-encoding of the attribute. For a table with m attributes, the feature vector has $4 \cdot m$ entries. For our table `R` follows $m = 3$ and a query with predicates `A < 25 AND B = 10`, the query featurization looks like:

$$\overbrace{\underbrace{000\,0.0}_{\text{no pred.}}}^{\texttt{id}}\overbrace{\underbrace{001}_{<}\,\underbrace{0.22}_{25}}^{\texttt{A}}\overbrace{\underbrace{100}_{=}\,\underbrace{0.85}_{10}}^{\texttt{B}}$$

All entries are padded with 0 for attributes for which a query contains no predicate, i.e., `id`. However, with SPE, multiple predicates on a single attribute cause problems because the encoding only supports one predicate per columns. Every additional predicate on the same column overrides the previous one. Therefore, SPE produces lossless featurizations for queries with up to one predicate per attribute and causes information loss for queries with multiple predicates per attribute. Disjunctions are not supported, so all predicates must be connected by `AND`.

Range Predicate Encoding

This section presents a straightforward but useful extension of SPE. Range Predicate Encoding (RPE) is a featurization that allows to encode, per attribute, either (1) a range predicate, where both open or closed ranges are supported, or (2) an equality predicate. Our predicate encoding technique builds on the observation that, in databases, all types of point and range predicates can be encoded to closed ranges. For instance, `A = 25` becomes $[25, 25]$ and `A` $\leq$ `25` becomes $[\min(\texttt{A}), 25]$. While the difference between range predicates including or excluding endpoints is often marginal, this can still be addressed. For integer attributes it is easy to see that `A < 25` corresponds to $[\min(\texttt{A}), 24]$ and for decimal attributes we can use a small step size, e.g. $[\min(\texttt{A}), 24.9]$. Since this is beneficial for ML models, all ranges are additionally normalized to $[0, 1]$ using the min and max values of each attribute. This shortens the vector representation for m attributes from $4 \cdot m$ values to $2 \cdot m$, one entry for the upper and lower boundary of the filtered range. Therefore, RPE only changes (3) of SPE to

(3) The predicate value range $[\min(\texttt{A}), 24]$ is min-max-normalized to $\frac{\min(\texttt{A})-\min(\texttt{A})}{\max(\texttt{A})-\min(\texttt{A})} = \frac{0-0}{115-0}$ and $\frac{\mathit{value}-1-\min(\texttt{A})}{\max(\texttt{A})-\min(\texttt{A})} = \frac{25-1-0}{115-0}$.

Every operator in $\{=, >, <, >=, <=, != \}$ has its own min-max-equation for lower and upper boundary. In addition, the need to encode the comparison operator (2) is eliminated. So, our example `A < 25 AND B = 10` looks like

$$\overbrace{\underbrace{0.0\,0.0}_{\text{no pred.}}}^{\texttt{id}}\;\overbrace{\underbrace{0.00}_{\min(\texttt{A})}\,\underbrace{0.20}_{25-1}}^{\texttt{A}}\;\overbrace{\underbrace{0.85}_{10}\,\underbrace{0.85}_{10}}^{\texttt{B}}$$

The benefit of Range Predicate Encoding is that all queries with up to one equality, open range, or closed range predicate per attribute are featurized losslessly. Queries with predicates defining multiple attribute ranges or range exclusions of single values are not supported. Neither are disjunctions, so all predicates must be connected by `AND`.

Universal Conjunction Encoding

The featurizations discussed thus far could be derived directly from the query but share one common disadvantage. They only encode a limited number of predicates in the feature vector without information loss. The problem is inherent to the previous featurization techniques because SQL queries are of arbitrary length but feature vectors have a fixed length.

Universal Conjunction Encoding (UCE) builds on the observation that both the number of attributes in the data set, denoted by m, and the data domain of each attribute remains constant within a context. Hence, while the same attribute may occur in multiple predicates, we use the fact that no query can reference more than m distinct attributes in a context. The data-driven idea that follows is to (1) partition the data domain of each attribute, (2) give each partition an entry in the feature vector, and (3) assign a value to each entry that indicates whether the partition it represents satisfies the predicates in the query. This technique allows us to encode queries with arbitrarily many simple predicates connected via `AND`. To implement this idea, we partition the domain of each attribute `X` into $n_{\texttt{X}} = \min(n, \max(\texttt{X}) - \min(\texttt{X}) + 1)$ buckets, where n denotes some maximum number of buckets per attribute. The feature vector entry corresponding to the value $v \in X$ has zero-based index

$$\left\lfloor \frac{v - \min(\texttt{X})}{\max(\texttt{X}) - \min(\texttt{X}) + 1} \cdot n_{\texttt{X}} \right\rfloor$$

Each feature vector entry indicates via a categorical value whether the corresponding bucket qualifies for the predicates in the query. We use 0 to indicate that the bucket qualifies, $\frac{1}{2}$ to indicate that parts of the bucket qualify, and 1 to indicate that the whole bucket qualifies. The concatenation of the per-attribute featurization yields the total feature vector.

The optimal choice of the maximum number of buckets n depends on the frequency distribution of the values in the m attributes. In general, we observe that each bucket covers $\frac{1}{n}$ of the domain of attribute `X`. For example, with n=8, each bucket covers roughly 12% of an attribute's domain. In the context of query optimization, this granularity has to be calibrated experimentally. Indeed, the evaluation in [63] supports $n = 32$ as a reasonable heuristic. For attributes with high skew or other non-uniform distributions, a larger n may be necessary.

Given the complexity of this featurization, we illustrate its workings with an example. Let the example table be `R` with $m = 3$ attributes `id`, `A`, and `B`. Let $n = 12$ be the maximum per-attribute buckets. Then, a query with predicates `id < 16 AND A >= 30 AND A <= 100 AND A != 66` induces the following feature vector:

$$\underbrace{111\frac{1}{2}00000000\,0.30}_{\texttt{id < 16}}\ \underbrace{000\frac{1}{2}11\ \overbrace{\frac{1}{2}}^{\texttt{A != 66}}\ 111\frac{1}{2}0\,0.60}_{\texttt{30 <= A <= 100} \wedge \texttt{A != 66}}\ \underbrace{11\,1.0}_{\text{no pred.}}$$

To stabilize the representation of the bucket-wise featurizations, we summarize the coverage of a predicate over all buckets with *per-attribute selectivity estimates*. These are represented by the gray values in our example featurizations. The per-attribute selectivity estimate of some attribute `X` is the ratio of `X`'s domain that qualifies by the predicates on `X`. This estimate is obtained as the part of the domain of `X` that qualifies under the predicate divided by the total range of `X`. This corresponds to an estimate under uniformity assumption like in [79].

With respect to `id < 16`, 16 maps to the fourth entry in the vector of `id` since, according to the zero-based index formula, $\lfloor(16-1)/(50-1+1)\cdot 12\rfloor = 3$. This fourth entry is set to $\frac{1}{2}$ because this bucket contains the values from 14 to 17 and only partially qualifies for the predicate. All entries to the left are set to 1 to indicate that values smaller than 16 qualify. Accordingly, all entries to the right, but within the bounds of `A`'s domain, are set to 0. Similarly, we introduce the predicate buckets for `A`. The closed range for `A` from 30 to 100 is mapped to the buckets with ids 3 and 10, both partially qualifying. The `B != 66` reduces the fully qualified bucket 6 to partially qualified because the value 66 is excluded from the range. Since there is no predicate on attribute `B`, and its data domain consists of only two values, `B`'s featurization is the all-one vector 11. The per-attribute feature vectors of `id` and `A` have 12 non-gray entries each, since this example assumes that $n = 12$ buckets in each per-attribute vector and that the domain size of `id` and `A` is greater than or equal to 12. The last attribute `B`, for which the example query contains no predicate, has only a domain size of 2. Thus, its number of buckets must also be 2.

Strictly speaking, there are no queries that can be featurized losslessly with UCE, unless there are more feature vector entries than distinct values for an attribute. However, UCE converges meaning that as the number of buckets n is increased beyond a certain level, the feature vector does not change anymore. Below this level, each increase in n reduces the information loss. Hence, for large feature vectors where each entry corresponds only to one distinct value of an attribute, UCE is a lossless query featurization for all queries with arbitrary conjunctions of simple predicates. For smaller feature vectors, this featurization loses only information up to the size of the attribute buckets that a feature vector entry represents. However, UCE cannot handle disjunctions.

Limited Disjunction Encoding

Limited Disjunction Encoding (LDE) is the first featurization that is designed to take both conjunctions and disjunctions, i.e. predicates connected by `AND` as well as `OR`, into account. LDE is a generalization of UCE. Before we present LDE, we address its limitations. The name already suggests that we cannot handle arbitrary disjunctions. We restrict ourselves to queries with conjunctions of compounds. A compound predicate is made of arbitrary many combinations of predicates for exactly one attribute connected with `AND` or `OR`. These mixed queries do not have to follow a conjunctive normal form or disjunctive normal form.

As for this class of queries, disjunctions occur only locally, i.e., per attribute. The key idea of LDE is to regard each conjunction in each compound predicate as a query that can be featurized using UCE. Therefore, for each compound predicate, the per-conjunction featurizations can be merged over disjunctions by taking the entry-wise max over all per-conjunction featurizations. This merging technique captures the property that additional disjunctions make queries only less selective. As for UCE, the final feature vector is the concatenation of all per-attribute entries.

To illustrate the idea, we featurize the example `(id > 2 AND id <= 30 AND id != 7 OR id >= 42) AND A > 39`. These filters are applied on our table `R` with attributes `id`, `A`, and `B`, with the min and max values from the example for the previous featurization. Suppose the attributes `id`, `A`, and `B` have the min and max values from the example in the previous section. The number of buckets is $n = 12$. The compound predicate on `id` consists of two conjunctions. For each conjunction, a featurized representation is generated using LDE. In particular, `id > 2 AND id <= 30 AND id != 7` is featurized to

$$\underbrace{\frac{1}{2}\quad\overbrace{\frac{1}{2}}^{\texttt{id != 7}}\quad 1111\frac{1}{2}00000\textcolor{gray}{0.54}}_{\texttt{2 < id <= 30}\,\land\,\texttt{id != 7}}$$

and `id >= 42` is featurized to

$$\underbrace{000000000\frac{1}{2}110.18}_{\texttt{id >= 42}}$$

The above per-conjunction vectors must then be merged by taking the entry-wise maximum. The per-attribute selectivity estimates in gray have to be recalculated to fit the new bucket distribution.

$$\underbrace{\frac{1}{2}\frac{1}{2}1111\frac{1}{2}00\frac{1}{2}110.72}_{\texttt{2 < id <= 30} \wedge \texttt{id != 7} \vee \texttt{id >= 42}}$$

The compound predicate on `A` only consists of `A > 39`, which is regarded as one conjunction. The featurized representation of this single predicate is

$$\underbrace{0000\frac{1}{2}11111110.66}_{\texttt{A > 39}}$$

Since attribute `B` is not mentioned in the query, its featurization is the all-one vector 111.0. The concatenation of the per-attribute vectors gives the final feature vector for the mixed query.

$$\underbrace{\frac{1}{2}\frac{1}{2}1111\frac{1}{2}00\frac{1}{2}110.72}_{\texttt{2 < id <= 30} \wedge \texttt{id != 7} \vee \texttt{id >= 42}} \quad \overbrace{0000\frac{1}{2}11111110.66}^{\texttt{A > 39}} \; \underbrace{111.0}_{\text{no pred.}}$$

Since merging feature vectors with maximums over disjunctions resembles the semantics of `OR`, and the single feature vectors to be merged converge to a lossless feature vector, it follows that LDE converges to a lossless query featurization of mixed queries.

Other Predicate Data Types

All previous query featurizations are introduced on integer data. However, all four techniques support other data types as predicate values. The implementation of floating point predicates is straightforward because all histogram-based calculations can be converted to floating point ranges [79]. String predicates are supported via a dictionary encoding in most other work to map strings to integers [93, 99]. This would only work for equality predicates. Range predicates are only supported for sorted dictionaries, and `X LIKE` *a*`%` predicates are not supported at all. The problem is that they cannot be encoded in the feature vector. However, UCE and LDE naturally support the encoding of such predicates. Consider, for example, a column where all entries are strings of lower case letters, i.e., $[a-z]^*$. With $n = 26$ entries in the per-attribute vector, each entry corresponds to the most significant letter of a word, e.g. words starting with d are represented by the fourth entry. This allows for any kind of predicate on a string-type attribute.

3.6 QUERY LABELING

One major point for optimizing **(A3)** regarding better training times of local models is the label generation for ML-based cardinality estimation, i.e, running queries against the DBMS to retrieve the true cardinalities, as shown in Figure 3.2. This applies to local and

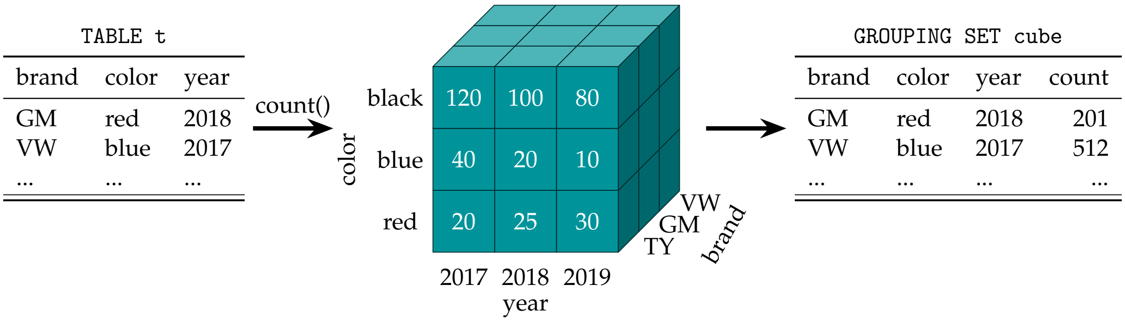


TABLE t

brand	color	year
GM	red	2018
VW	blue	2017
...	...	...

GROUPING SET cube

brand	color	year	count
GM	red	2018	201
VW	blue	2017	512
...	...	...	...

Figure 3.9: The aggregate-based labeling with grouping sets.

global learning strategies if they are supervised. The generation of labels puts a high load on the database because usually thousands of queries are required to be labeled for training. It is desirable to reduce this load as much as possible to not hinder user workloads also running against the database. Since the presented works for ML-based cardinality estimation are all supervised, we present an universal label generation technique that works for all supervised label retrieval. The presented work is derived from [91].

An important observation about the training workloads for local and global models is the fact that the queries are structured similarly [44, 93]. They access the same joined tables and corresponding predicates repeating the same patterns. For example, the following collection shows that training queries on a car data table use similar combinations of predicates.

```
SELECT count(*)
FROM t
WHERE brand=‘GM‘ AND color=‘red‘ AND year=2018;

SELECT count(*)
FROM t
WHERE brand!=‘VW‘ AND color=‘blue‘ AND year=2017;

SELECT count(*)
FROM t
WHERE brand=‘GM‘ AND color=‘red‘ AND year<2018;
```

The set of predicates differs in its predicate values and operators, but the queries access the same table and the same columns. This is a general observation from several training workloads in the related work with the only extension that most of the time joined tables are accessed and not single tables. However, for these tables the observations still hold. In the following, we present an aggregate-based labeling and training phase that allows to speed up similarly shaped workloads.

To tackle this potential systematically, the core idea is to pre-aggregate the base data for different predicate combinations and to reuse this pre-aggregated data for several training queries. Aggregates compress the data by summarizing information and reducing redundancy. This lessens the amount of data to be scanned by each training query because the aggregates can be smaller than the original data. Therefore, it is also important that the construction of the aggregate does not take longer than the reduction of the workload execution time.

DBMS already offer substantial supportive data structures for aggregation. The most common ones of such groupings come from OLAP workloads. These workloads require the pre-aggregation of information in data cubes helping to reduce the execution time of OLAP queries [4, 34]. Each attribute of a table or join generates a dimension in the data cube and the distinct attribute values are the dimension values. The cells of a data cube are called facts and contain the aggregate for a particular combination of attribute values. A common operator for cubes in databases is the GROUPING SET. With the grouping set aggregation, we compress the original data and avoid the calculation of unnecessary attribute combinations. Given the grouping set data structure, we adapt the generation of labels by adding an intermediate step before executing the workload for cardinality retrieval as shown in Figure 3.2. This step constructs pre-aggregates for the data as shown in Figure 3.9. With a count(*) aggregate and the grouping set, a cube is initialized over all possible columns in a first step. For our example the CREATE statement for the grouping set is:

```
CREATE TABLE cube AS
(SELECT brand, color, year, count(*)
 FROM t
 GROUP BY GROUPING SETS((brand, color, year)));
```

This creates the grouping set cube as shown in the figure. In a second step the queries need to be rewritten to match the new data structure.

```
SELECT count
FROM cube
WHERE brand=‘GM‘ AND color=‘red‘ AND year=2018;

SELECT count
FROM cube
WHERE brand!=‘VW‘ AND color=‘blue‘ AND year=2017;

SELECT count
FROM cube
WHERE brand=‘GM‘ AND color=‘red‘ AND year<2018;
```

We only need to modify the projection from a count(*) aggregate to directly use the count and replace the table with the corresponding grouping set. The predicate structure is kept by the aggregation and needs no rewrite. A corresponding grouping set can be initiated over every model context, i.e., a set of joined tables (cf. Section 3.3)

Even though our approach is independent of the ML model, it is not independent of the data. In general, grouping sets (GS) are only beneficial if the aggregate is smaller than the original data. To quantify this prerequisite, we define a *benefit criterion*. It is defined over the number of columns C, the number of tuples N, and the distinct values of a column dv in a certain model context.

$$\text{benefit} = \frac{1}{N} \prod_{c \in C} dv(c) \tag{3.13}$$

If the benefit criterion is smaller than one for a model context, a grouping set is constructed. The aggregated data in the grouping set will be smaller than the original data and the runtimes for scanning will be faster. This criterion also details the amount of data we save with the aggregate. Therefore, it is a heuristic for the performance scaling.

3.7 EVALUATION

In this evaluation, we show the advantages **(A1)** to **(A4)** of our local strategy for cardinality estimation. We detail the positive influence with several experiments regarding **(A1)** less complex models, **(A2)** better estimation quality, **(A3)** faster training, and **(A4)** faster forward passes.

3.7.1 Setup

For the following experiments, we use two real-world data sets together with corresponding query workloads. The first data set, forest cover type (forest) [52], is popular both in the machine learning and cardinality estimation community and contains more than 580k entries with 55 attributes. For forest, we generate a query workload with conjunctive queries, i.e., predicates connected by `AND`. We draw $k \in [1, 55]$ distinct attributes uniformly at random and randomly generate a closed range predicate for each. Additionally, we generate $l \in [0, 5]$ not-equal predicates for each of the k chosen attributes, excluding values from the range mentioned above, where l is drawn uniformly at random. For instance, one of the queries from our evaluation is

```
SELECT count(*)
FROM forest
WHERE A7 >= 160 AND A7 <= 225 AND A8 >= 45
AND A8 <= 237 AND A8 <> 220 AND A8 <> 186;
```

In addition, we generate a second query workload with mixed queries containing conjunctions and disjunctions. The generation is the same as for conjunctive queries, except that we repeat the generation for the per-attribute predicates between $m \in [1, 3]$ times and concatenate them via `OR`. For both conjunctive and mixed queries, we generated 100k training queries and another 25k test queries.

As a second data set, the Internet Movie Database (IMDb) [49] is used. IMDb contains data on over 2.5 million movies with around 4 million actors from more than 135 years. For testing, we use JOB-light, a collection of 70 hand-written analytical SQL queries from [44]. The JOB-light queries contain between two and five joins. The selection predicates are only conjunctions of one to five predicates on one to four different attributes. The queries contain, at most, one range per attribute. For training, 231k generated training queries are used. In all query workloads, since training and test sets are disjoint, test set leakage is avoided.

For the NN, we use the Keras/TensorFlow implementation provided by [93]. The GB models are built with scikit-learn. MSCN and its modifications are built upon the code published alongside the paper [44, 43]. We ran all experiments on an AMD A10-7870K Radeon R7 machine with 32 GB memory and an NVIDIA Tesla K20c. The neural networks are trained only with the hyperparameters from their papers due to long training times. The GB models, on the other hand, are trained with full hyperparameter tuning, implying that the presented results are based on the best configurations.

Table 3.3: Model complexities for JOB-light.

Model	Parameters per model	Parameters total	Memory per model [kB]	Memory total [kB]
MSCN	$\approx$ 2,600,000	$\approx$ 2,600,000	320	320
local NN	$\approx$ 130,000	$\approx$ 2,300,000	15.7	283
local GB	$\approx$ 1,700	$\approx$ 31,000	0.27	4.8

Table 3.4: Q-error for different models and featurizations on JOB-light.

Model + featurization	Mean	Median	99%	Max
PostgreSQL	174.01	7.88	218.56	3477.20
DeepDB	6.08	1.66	55.20	69.49
NN + SPE	144.47	10.67	2507.34	3331.07
NN + RPE	110.23	7.60	2050.50	3573.30
NN + UCE	19.97	5.74	129.45	134.37
GB + SPE	4.03	1.88	34.06	56.39
GB + RPE	**3.92**	1.65	**29.77**	**45.51**
GB + UCE	8.88	**1.52**	106.10	114.55
MSCN + SPE (original)	138.9	11.23	4209	5460
MSCN + RPE	133.8	5.55	2311	2861
MSCN + UCE	119.8	5.26	1465	1811

3.7.2 Model Complexity

First, model complexity is evaluated. Assessing the complexity of a model can be done in two ways: via the number of model parameters or the model's memory consumption. The desired outcome is to have models with only few model parameters or a small memory footprint. To show the advantage of a local approach, we compare the MSCN [44] from related work with two possible modes for local models. These are feed-forward neural networks (NN) and gradient boosting models (GB) as described in Section 3.4.

Table 3.3 shows the number of parameters and the memory footprint of all three models for estimating the Join order benchmark (JOB)-light benchmark [44]. The numbers are reported for both a single model and the total complexity for all necessary models, especially for the local approach because it requires more than one model to cover the problem space. The JOB-light benchmark contains 70 queries spanning 18 contexts on the IMDb data set limited to integer-only attributes and predicates. The complexity for a single MSCN covering the whole schema is higher, both in parameters and memory, than the 18 smaller NN models covering the 18 contexts within the JOB-light. However, compared to GB models their complexity is factor 83 higher for the parameters and factor 66 higher for memory. Assumed that these models produce the same quality (or even better) in estimates then it is clear that the model independence of our local strategy can reduce the model complexity. This is reached by using small local models that are able to model the reduced problem space of context with fewer parameters. Therefore, the collections of model is as expressive as the global approach, but uses less memory. So, advantage **(A1)**, lesser model complexity, can be achieved with a local learning strategy.

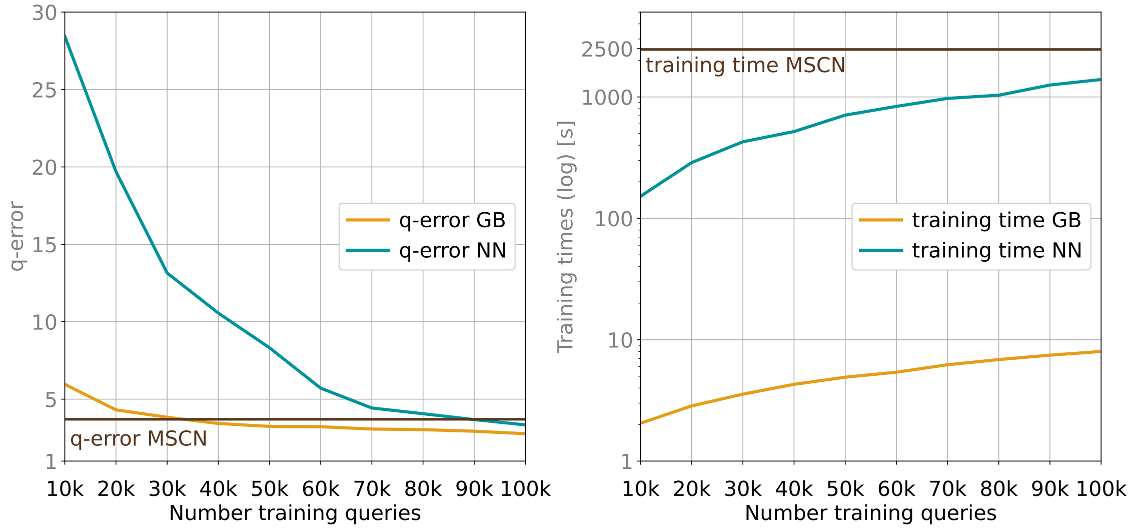


Figure 3.10: The trade-off between quality and training times according to number of training queries.

3.7.3 Model Quality

To show the positive impact of the local model strategy on estimation quality **(A2)**, we show both different model types and different featurizations and their improvement to the process. We evaluate all techniques on the JOB-light, but this time we look at the qualities rather than the model complexities. Training data was generated according to these 70 queries but test set leakage has been avoided. The used error metric is the *q-error* [61], which details the factor of misestimate in cardinality between the true cardinality and the estimate and is defined as:

$$\text{q-error}(true, estimate) = \max\left(\frac{estimate}{true}, \frac{true}{estimate}\right) \tag{3.14}$$

Table 3.4 shows the different models local feed-forward neural networks (NN), local gradient boosting models (GB), and the MSCN from [44] under different featurizations, except LDE because JOB-light has no disjunctive predicates. We also include the PostgreSQL estimator and DeepDB [36] as baseline comparisons. In the table, the statistics mean, median, 99% percentile, and maximum of the q-error of the estimates for the workload are reported. Best values have been highlighted in bold font. A general observation is that more complex featurizations improve the quality of all model types. RPE outperforms other featurizations with the local strategy because closed range predicates are the most complex ones in this workload. Additionally, all local models are almost always better than the global MSCN throughout all reported statistical numbers. So, we show two things at once with this experiment: (1) the modeling of queries with featurization is important for estimation quality and (2) using local learning strategies is beneficial for cardinality estimation quality. Furthermore, it shows that model independence as proposed in Section 3.4 is given for our local strategy because we can use any of the two NN or GB models. However, for this particular workload, GB produces the better results, i.e., estimates. Therefore, **(A2)**, better qualities, can be reached with a local learning strategy.

Table 3.5: Times of forward passes.

Model	Forward pass per query [µs]	Forward pass total [ms]
MSCN	33,000	2,310.00
local NN	29	2.03
local GB	26	1.82

3.7.4 Training Time and Forward Pass

Next, the advantageous influence of the local strategy on training times **(A3)** and forwarded passes **(A4)** is evaluated. This is referred to as the *sample efficiency*. An ML model is more efficient, the less data it needs to produce high quality estimates. For the experimental setup, we are using the Forest CoverType data set [52]. On this data set, 125k complex queries are generated. 25k queries are put into the test data set, whereas the number of training queries is increased step-wise. For every step, the model estimation quality (q-error) on the fixed test set is calculated. We focus on GB+UCE and NN+UCE compared to MSCN without modifications. The MSCN is only used as a maximal threshold for both training time and q-error. To directly compare the models' performances, we only instantiate one context and disable hyperparameter tuning. The results are shown in Figure 3.10. Looking at the quality, i.e., the q-error, on the left side, we observe that the GB models reach the model quality of MSCN with 40k training queries, whereas the NN requires 90k queries. The training times for both local models is much lower than for the MSCN. In the log-scaled right plot the training time for GB with 40k queries is a little over four seconds and for NN with 90k queries it is around 1,200 seconds. The MSCN takes 2,500 seconds in training. The local model strategy can reach the same quality as a global approach with faster training by factors 570 (GB) and 2 (NN). Therefore, the local approach has a higher sample efficiency than the global approach. We argue that GB models have an advantage over NN because they are even faster in training and reach competitive qualities earlier. This shows that **(A3)**, faster training, can be shown for a local learning strategy in cardinality estimation.

Lastly, we compare the forward passes of different local and global models to verify that the local strategy fulfills **(A4)**, faster forward passes. The forward pass is crucial to be fast because it is executed for every query to be estimated. So, the faster the forward pass through a model, the smaller the overhead during model application and query optimization. Table 3.5 shows the forward passes of MSCN, local NN, and local GB for single queries and the complete JOB-light workload. For both single-query and full workload performance, the local models are factor 1,000 faster. This also means that the application of local models is orders of faster during query optimization and runtime. Therefore, **(A4)** can be derived as an advantage of the local learning strategy.

3.7.5 Query Labeling

To evaluate the influence of our aggregate-based labeling, we compare its performance on the training workload of the MSCN (global) [44] and three different workloads for local models (local 1 to 3) [93]. All workloads are artificially generated and based on the IMDb data set. For the global workload, we run two experiments: *global full* shows the performance of the construction of all possible cubes without employing the benefit criterion. *Global opt* only contains the constructions of beneficial GS according to the benefit criterion. The three local workloads build upon each other. The local 1 workload

Table 3.6: Execution times for training workloads with grouping sets (GS).

model	Base data w/ index	Construction GS	Execution GS	Total GS	Total GS w/ index	Coverage GS
local 1	$1h\,44m$	$6.17s$	$191.34s$	$197.51s$	$139.47s$	100%
local 2	$5h\,10m$	$23.70s$	$205.91s$	$229.61s$	$149.99s$	100%
local 3	$6h\,56m$	$29.10s$	$430.36s$	$459.46s$	$295.25s$	100%
global full	$4d\,14h$	$2h\,22m$	$20d\,20h$	$20d\,22h$	–	100%
global opt	$4d\,14h$	$34m\,29s$	$2d\,11h$	$2d\,12h$	$1d\,21h$	55%

contains training queries for one context, local 2 for two contexts, and local 3 for three contexts, where the contexts from the previous workload are always included in the next one.

Table 3.6 details all results for the five experiments. We compare the baseline labeling execution with indexes, the total labeling runtimes with GS (additionally split into GS construction and workload execution), the labeling runtime of GS with additional indexes. Additionally, we show the coverage of the GS meaning how many queries of the workload are rewritten to use GS. The experiments show that a GS-based pre-aggregation can improve the labeling process for both global and local models by factors up to 80. It also shows that initiating all GS for a workload without a heuristic can worsen the labeling runtime enlarging the construction and access times because the GS become larger than the original data. Therefore, two things can be derived: (1) using GS is beneficial for faster labeling of queries and (2) GS have to be placed according to the benefit criterion to avoid unnecessary overhead. Besides that, the evaluation shows again that using a local learning approach has another advantage because it can use full potential of GS-based labeling. This leads to even faster training times for local models according to **(A3)**.

3.8 SUMMARY

From this chapter, it is clear that the local learned selection strategy for cardinality estimation offers the four advantages as defined in Section 3.3 compared to global models.

- **(A1)** Lesser model complexity by specializing to a sub-problem, i.e., a sub-part of the schema
- **(A2)** Better qualities with more expressive local models focusing on predicate combinations
- **(A3)** Faster training times by using fewer training queries and grouping sets for labeling
- **(A4)** Faster forward passes from less complex local models

Additionally, our aggregate-based labeling strategy improves the **(A3)** training times even further. This allows the conclusion that local models both improve the quality and performance of ML-based cardinality estimation, even though the construction of several models seems counter-productive at first. The limited complexity of the sub-problem spaces requires only models that are in sum better and faster than one large global model. We have shown this extensively throughout the chapter. We were also able to include our approach into the open-source database system PostgreSQL [94]. There, we can show that the four advantages also hold in a real-world environment. However, we notice that the PostgreSQL optimizer is hard to influence with cardinality estimates, even if they are close to the true values [63, 94]. This is due to the defensive nature of the optimizer towards misestimates because it needs to account for large estimation errors to avoid catastrophic query runtimes. Nevertheless, a local learning strategy is generally beneficial for ML-based cardinality estimation regarding quality and performance.

4

THE LOCAL LEARNING STRATEGY FOR QUERY OPTIMIZER HINTING

The increasing amount of data managed by DBMSs, supports the argument that efficient query processing still is a critical challenge [18]. To tackle the optimization task behind the problem, every DBMS contains a query compiler that converts each incoming SQL query into a *query execution plan* (QEP) [42]. This includes the *query optimizer*, which should determine the most efficient QEP [16]. Despite much research activity, query optimization is still far from being solved and, therefore, lacks efficient plan selection in some cases [17, 50]. According to [16], the most challenging issues for the optimization of complex analytical queries in the form of select-project-join (SPJ) queries are (i) finding a good join order and (ii) selecting the best-fitting physical join implementation for each join within the chosen join order. To solve these challenges, traditional query optimizers use three components: (i) an enumerator that spans the search space of all possible QEPs, (ii) a cost model to assess the cost of any given QEP prior to its execution, and (iii) a cardinality estimator delivering table sizes to the cost model. Figure 4.1 shows this structure and its interactions. The role of cardinality estimation is described in more detail in Chapter 3. In this chapter, we focus on the steering of the whole query optimizer with hints to produce better query performance.

Traditional query optimizers in many systems, e.g., PostgreSQL [73], MySQL [64], Oracle [68], or SQL Server [80], feature a rich set of configuration parameters to influence the properties and choice of the selected QEP. These configuration parameters are also known as *query optimizer hints*. Furthermore, a one-elementary configuration is referred to as a *hint*, a multi-elementary configuration as a *hint set*, and the process of configuring and applying hints or hint sets as *query optimizer hinting* or just *hinting*. As detailed in Figure 4.1, the hints can influence any part of the query optimizer. For example, the use of certain physical join operators can be enabled or disabled in PostgreSQL [73]. Setting such a hint for a query results in the underlying query optimizer to either consider or strictly avoid certain QEP techniques, like disabling nested loop joins in Figure 4.2. Therefore, the optimizer can be hinted externally on a single query basis. Hinting is also called *optimizer steering*.

4.1 PRELIMINARIES FOR LEARNED QUERY OPTIMIZER HINTING

As shown in [35], physical operator selection significantly impacts analytical query optimization. Therefore, we focus on the six primary Boolean hints for physical query optimization in PostgreSQL in this thesis. These hints contain annotations to turn on or off certain scan (sequential, index, index-only) and join (hash-join, merge-join, nested-loop)

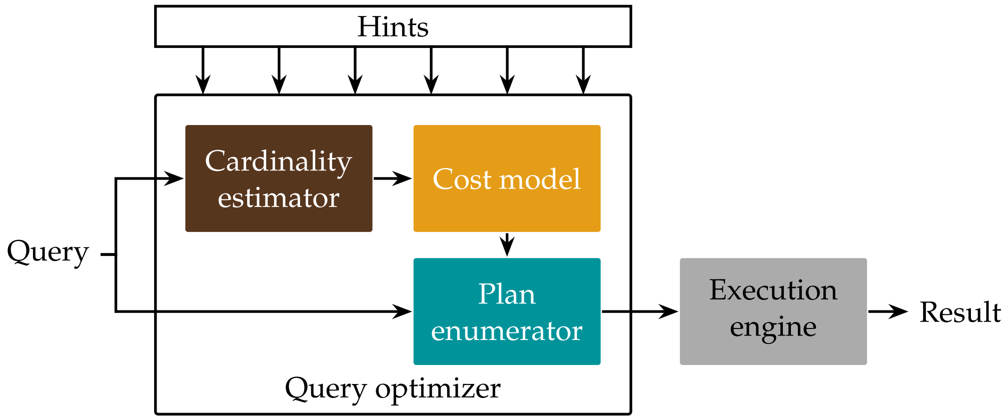


Figure 4.1: The structure of a traditional query optimizer with hints.

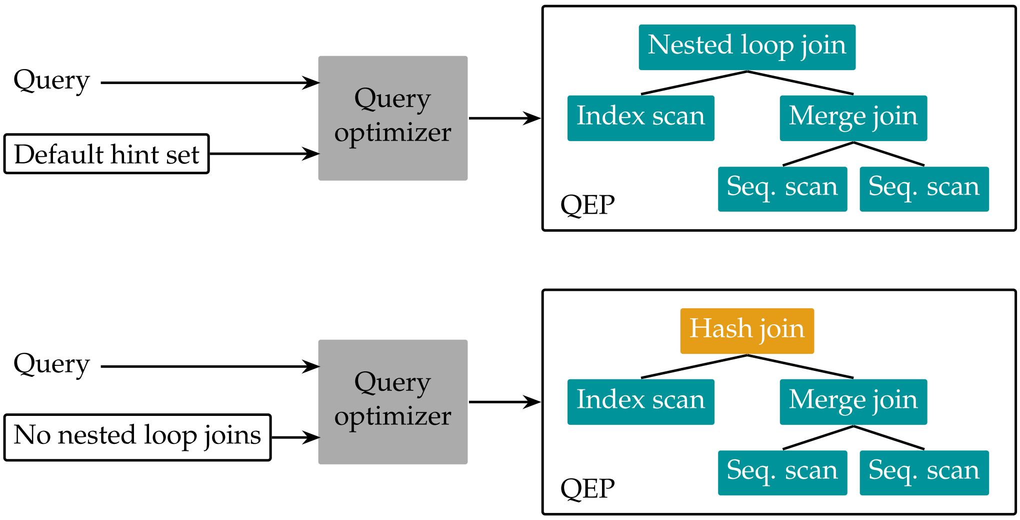


Figure 4.2: The QEP changes if nested loop joins are prohibited via hints.

operations. We represent a hint set in two ways: a bit vector and an integer. The bit vector shows the status of each of the six Boolean hints in the hint set. The integer representation is the direct transformation from bit vector to integer. For example, the hint set enabling only the joins via the first three hints is $111000_2 = 56_{10}$. In the default PostgreSQL setting, all six configuration parameters are enabled to span the largest possible plan search space. This hint set is called *PostgreSQL default* and is labeled with $111111_2 = 63_{10}$.

In particular, we want to use a local learning strategy explicitly designed to augment any relational cost-based query optimizer that can be steered by hints. The local learning strategy should directly predict hints for a query. Subsequently, a traditional cost-based query optimizer is invoked using the query and the predicted hints. We consider the underlying query optimizer a black box, facilitating broad applicability. Like in the previous chapter, we do not learn one global model but a distinct model per well-defined sub-problem to make fine-grained predictions of hint combinations on a query level. Results and parts of this chapter are based on [95].

In the following Section 4.2, we present related work on query compiler hinting, mainly with global models, and their shortcomings and derive four advantages that our local approach will have. Next, we introduce our local learning strategy for query compiler hinting in Section 4.3. To even further improve the capabilities of our local learning strategy during runtime, we present a smart labeling strategy in Section 4.4. In Section 4.5, this is complemented by the definition of a process to avoid outdated models during strategy application. We evaluate the advantages of our approach in Section 4.6. Lastly, we summarize the finding of this chapter in Section 4.7.

4.2 RELATED WORK

Direct query optimizer hinting is a relatively new area in database research [54, 95]. However, query optimization of SQL queries, especially analytical queries, has been a research topic for decades, but it is still not solved [17, 49]. So, most related work does not focus on direct hinting but on classical components of query optimization. For holistic query optimization, one can address any of the three query optimizer components: cardinality estimation, plan enumerating, or the cost model. The following sections will detail related work from these three areas. Lastly, we present the limited related work in direct hint prediction.

4.2.1 Cardinality Estimation

As presented in Chapter 5, optimizing queries depends on the accuracy of cardinality estimates, particularly for intermediate results sizes [16]. However, traditional cardinality estimation techniques frequently rely on basic heuristics that may assume predicate independence and uniform distribution of attribute values [49]. Therefore, the most apparent application area for ML in query optimization is cardinality estimation, as shown in Section 3.1. Most of the presented related approaches only show improved accuracy in cardinality estimation. However, they seldom show that improved accuracy leads to accelerated query performance. One exception is our work, which details several improvements in query runtimes [63, 94]. A more detailed overview of related work in cardinality estimation can be found in Section 3.2.

4.2.2 Plan Enumeration

For plan enumerating, other ML-based approaches use complex global end-to-end solutions. For example, Yang et al. try to avoid using expert planners in their approach called BALSA [98]. Here, partial query plans and their resulting latency are learned by a neural network in a two-stage manner. Firstly, a simulation phase utilizes an expert cardinality estimator (e.g., PostgreSQL's `EXPLAIN ANALYZE`) to initialize a QEP to cost mapping. The initial model is refined in the second stage using actual latency measures collected during runtime. The best possible join pattern for each new query is selected by choosing the best latency of join orders in a bottom-up fashion based on the predicted costs of the neural network. This facilitates a greedy join order selection on trained join patterns and their respectively predicted costs. However, the greedy bottom-up processing of joins can lead to sub-optimal query plans because the optimization can get stuck in a local optimum. This, in conclusion, can lead to sub-optimal query runtimes. Additionally, BALSA has long training times, making it unfeasible in a real-world scenario.

In contrast, Marcus et al. use a tree convolution network (TCN) with query and plan encodings as inputs in their model NEO [56]. The query encoding consists of the upper triangular adjacency matrix for join partners. This matrix enables mapping all possible join partners in a schema by setting the corresponding adjacency for all joined table pairs. The plan encoding is the output of an expert optimizer that generates a query plan tree. Again, PostgreSQL's `EXPLAIN ANALYZE` is a prominent example. The query encoding is forwarded to a fully connected neural network, which generates an output. This output is concatenated with the previously supplied plan encoding through the query plan tree. The concatenation result is then used as an input for the TCN that outputs a query execution time prediction. According to the lowest predicted execution time, the best query plan tree is executed in the execution engine. Training data is collected during query execution by storing query runtimes. While this TCN-based plan enumeration demonstrates improvements in mean query performance, the long training times hinder its practical application. Additionally, relying on an external optimizer only mimics its capabilities and might lead to overfitting [95]. Therefore, the model cannot choose the best plan because it is presented only with the sub-optimal plans of the external optimizer. This leads to sub-optimal query execution times.

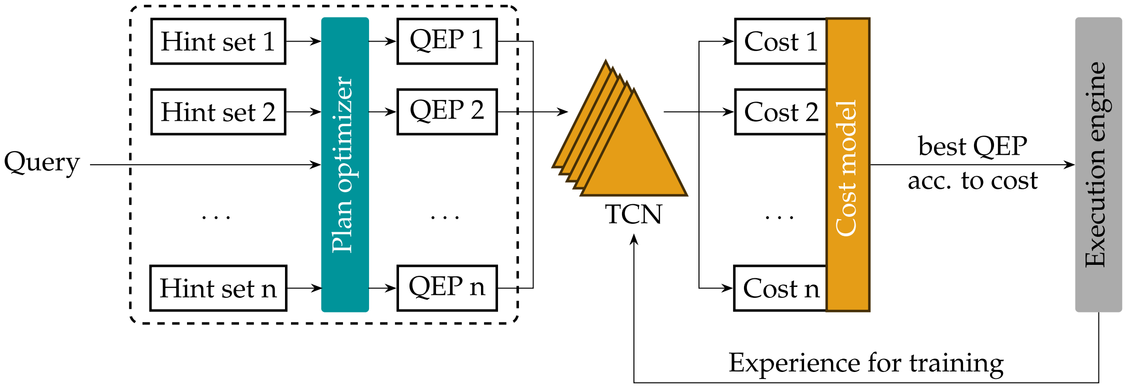


Figure 4.3: The Bandit Optimizer (BAO) for learned query optimizer hinting.

4.2.3 Cost Models

Regarding cost models, Hertzschuch et al. present a Case-Based-Reasoning approach called TONIC to enhance QEPs with learned physical join operators [35]. The input of TONIC is a QEP determined by a cost-based optimizer, while the output is a QEP with the same join order but with learned physical operator selections. For this purpose, TONIC collects, for each executed query, the QEP with a summary of the exact costs for the used operators in a case base. This is done for every node in the QEP. For each new incoming query, an external query optimizer determines the QEP, which is then retrofitted in a subsequent step by TONIC using a stored QEP from the case base. The retrofitting is the replacement of a physical operator on a node of the QEP. The replacement is based on the best stored costs for the particular node according to its position in the QEP and the used filter predicates. The case base structure is a prefix tree, which allows fast insertion of new data and fast matching for incoming queries. The authors show that TONIC accelerates the performance of queries and especially tail latency queries. However, the performance benefit with a cost-based optimizer is limited since the QEP, especially the join order, is not changed by TONIC. This is especially problematic when another join order would produce better query runtimes. TONIC is not capable of suggesting a better join order because it is fixed to cost-based physical operator optimization.

4.2.4 Direct Query Optimizer Hinting

The most recent and relevant ML approach for query optimization with hinting is the Bandit Optimizer (BAO) [54]. BAO is the first work to use hint-based query optimization, where the optimizer is used as a black box. Instead of directly optimizing the query optimizer components, it optimizes the external hints given to the optimizer. Thus, BAO is able to steer the whole optimizer without having to interfere with the details of its inner workings. Given the predicted hints by BAO, a traditional optimizer can work more effectively and improve query performance. Figure 4.3 demonstrates the overall process of BAO. An incoming query is forwarded to the plan optimizer under different hint sets. These hint sets are toggled on the fly and for each hint set, the optimizer is restarted with the same query. Here, the plan optimizer is the standard PostgreSQL optimizer called directly by BAO. The plan optimizer produces different QEPs, including join order and physical operators for different hint sets but the same query. All QEPs are forwarded to a tree convolution network (TCN), a specialized form of a neural network that is able to use QEP trees as input. The TCN predicts the costs, i.e., the runtimes, for all query QEPs, and, therefore, for all hint sets. These predicted costs form a new cost model from which the best QEP is chosen according to the lowest costs. The execution engine then not only runs the query but also collects the actual runtimes of the query. The experiences are saved as training data for the TCN during DBMS uptime. This makes BAO a reinforcement learning approach. Whereas BAO does not directly predict the hint sets, these can be derived from the best QEP and its corresponding hint set.

There is also related work by Yu et al. labeled as hint-based query plan optimization with a hybrid learned and cost-based optimizer [100]. However, this work has a different understanding of hints. Hints in this publication are prefix trees with partial join orders used as indicators for the final join order to be inferred by an ML model. Additionally, the model-based join ordering and plan selection are deeply integrated into the database system and not decoupled as in other work.

4.2.5 Shortcomings of Global Strategies

All presented ML-based approaches use global models. For example, BAO is a large complex model spanning a whole workload. This is comparable to the way that the MSCN predicts cardinalities. Over all queries, these models try to incorporate different properties over the whole problem space leading to the same four shortcomings as in the previous chapter for cardinality estimation. We will use BAO as a prime example of a global model and how its shortcomings are manifested because it is the first work in the area of learned query compiler hinting. However, the shortcomings are transferable to any other global model for learned query optimizer hinting.

(S1) Large complex models: The global models for query compiler hinting can become complex in structure to encompass as many properties of the queries as possible. Like cardinality estimation, the complexity is derived from the part of the schema the model needs to cover. Global models cover the whole schema and all combinations of tables and predicates. Therefore, their complexity is maximal according to schema characteristics. Even though BAO is based on trees, the TCN has a large memory footprint because it is still a complex NN [54]. Again, it is desirable to reduce this consumed memory for better DBMS performance.

(S2) Overfitting: The overfitting of global models including BAO stems from their all-at-once modeling. BAO is not able to solve the whole schema complexity, represented through QEPs because its structure does not allow for modeling finer differences between queries. Usually, it tries to average its prediction. This is sufficient for most queries, but tail queries with long runtimes suffer from this because they do not have average properties. Additionally, BAO ignores the hardware dependency of query execution [35, 95]. This also reduces the overall quality of hinting.

(S3) Amount of training data: As shown in the previous chapter, global models need a lot of training data to reach acceptable qualities. In query hinting, the training data are workload queries, just like in cardinality estimation. BAO uses a reinforcement learning approach that learns continuously during runtime. This leads to two disadvantages. The first disadvantage is that BAO produces catastrophic hint set predictions and long query runtimes for the batch of queries. The authors of BAO circumvent this problem by turning off BAO's prediction for the first 20 queries and only using them for training [54]. This basically makes BAO a supervised asynchronous approach during this time and diminishes the proposed advantages of a reinforcement approach, like fast adaptability to workload changes. The second disadvantage is the high amount of time needed before reaching good-quality predictions. We have shown that BAO needs the same amount of time, or even more, to reach comparable prediction quality to a supervised offline approach [95]. We will show that our local approach outperforms BAO even with only 10%

of all data as training data. Additionally, the supervised approach does not interfere with the DBMS during runtime. A reinforcement approach might introduce blocking behavior to query processing, which is undesirable.

(S4) Slow forward passes: The prediction of the costs per QEP is costly for BAO because of two aspects. Firstly, the general complex structure of the TCN with a lot of weights and calculations leads to slow forward passes for the model itself. This is the same for any complex NN as a global model in any use case. The second aspect is specific to BAO's architecture. The repeated calling of the optimizer under different hint sets is very compute-expensive. Every single call adds to the overhead of the process. It is also not comparable to the traditional process of the plan optimizer because the optimizer does a whole plan optimization to find the best QEP for every hint set. So, calling the optimizer twice in BAO is the same as calling the optimizer on two different queries in a traditional scenario. This is a considerable overhead acknowledged by the authors [54]. Their solution is to limit the number of different hint sets to five. This very much limits the applicability of BAO because it is not shown all possible solutions.

In the next section, we introduce our *local learning strategy for query optimizer hinting* that supports query optimization by predicting hint sets for the query optimizer. Just like for cardinality estimation, we will derive the four advantages tackling the four shortcomings **(S1)** to **(S4)**. In the following two sections, we present two additional features of our local strategy that make local models more efficient and competitive in applications. In Section 4.4, a *smart labeling* process is introduced to reduce the overhead of generating training data and labels during runtime. To avoid the degeneration of the local models becoming stale during workload or data shifts, we define a *runtime model maintenance* in Section 4.5.

4.3 THE LOCAL LEARNING STRATEGY FOR QUERY OPTIMIZER HINTING

To bring the local divide-and-conquer approach to query optimizer hinting, we build upon the findings of the local learning strategy for cardinality estimation. Since hinting is also, like cardinality estimation, very workload-dependent [54], we use the same definition of a context as for cardinality estimation in Chapter 3. To reiterate, we define a *context* in a local learning strategy for query compiler hinting as the accessed and joined tables in a query. Again, this stems from dividing the expressiveness of a query into predicate and join properties. The idea behind our approach is that each set of joined tables represents a self-contained context since the queries per context are reasonably homogeneous with respect to the joins and differ only in the filter predicates. Whereas the predicates are mapped by a featurization, the join properties are covered by partitioning the workload queries into contexts. A thorough description and argumentation of contexts can be found in Section 3.3. As a new technique, contexts can also be created on the fly, making the local learning strategy for query optimizer hinting adaptable to workload changes. If a new query arrives, it is either mapped to an existing context or a new context is created for the query. Due to the simple structure of a local model, it is possible to obtain an expressive model for a context with only a few training queries. This is a major advantage compared to global models where setting up a new model takes much longer.

So, given a workload with queries belonging to different contexts, we build a local model for each context to predict hint sets for these queries directly. This makes our local learning strategy a *multi-class classification* because the direct prediction characteristic requires the model to predict a vector containing every value for each hint in the hint set at once. This derives a general benefit of the local strategy. It is capable of *directly* predicting *all*

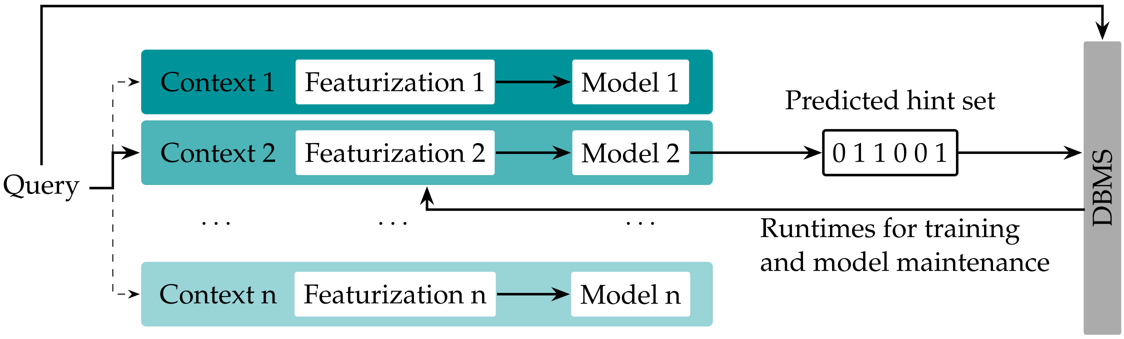


Figure 4.4: The local learning strategy for query optimizer hinting.

hints at once without any layers of indirection. Additionally, the local process or component can be kept simple compared to global strategies. Figure 4.4 details the overall process of the *local learning strategy for query optimizer hinting*. An incoming analytical query is mapped to a specific context according to the tables joined in the query. Every context contains a specialized featurization picking up the different predicates within each context, i.e., columns in the joined tables. Until this point, the process is similar to the approach for the local learning strategy for cardinality estimation. We decided to keep the featurization as simple as possible to limit the resulting feature vectors' length. Therefore, we use the Singular Predicate Encoding as presented in Section 3.5 and encode the operators in a one-hot vector and predicate values in the min-max range of the column. These featurizations are forwarded to the context model. Whereas we are flexible in the choice of local models, we choose gradient boosting (GB) models because of their favorable properties, as demonstrated in the previous chapter (cf. Section 3.4). However, the local approach allows any model to be used for local models. Due to the multi-class nature of the GB classification models, the context model can directly predict a vector containing the values for the whole hint set over a defined number of hints. In our case, a hint set is comprised of six binary hints as defined in Section 4.1. The predicted hint set is used to configure the DBMS, after which the query is executed in the configured DBMS. Through this configuring, the local learning approach steers the DBMS without interacting with its inner architecture. This is contrary to the approach BAO uses, which requires deep integration into the DBMS. It also allows our approach to be transferred to any underlying DBMS implementation. Another point avoiding a deep integration is the *supervised* character of our approach. Unlike BAO, a *reinforcement* learning approach, the local learning strategy does not need to interact with a DBMS during training. It can be trained beforehand and then easily deployed without the required overhead of a reinforcement approach.

Advantages of the Local Learning Strategy

With the definition of our local learning strategy for query compiler hinting, we can derive the same four advantages **(A1)** to **(A4)** as for the local strategy for cardinality estimation. These four advantages define the core properties of the local selection strategy for query optimizer hinting and will be evaluated in Section 4.6.

(A1) Less complex models: With the division of the problem space into contexts, each local model solves a sub-problem. This leads to smaller models that can solve the same task as one large global model. For hint set prediction, we can show that lightweight GB models are sufficient to classify the best hint set for an incoming query. Compared to BAO, the local learning strategy also does not need the plan optimizer as an additional component. This reduces the overall complexity of the local strategy because it needs one very complex component fewer.

(A2) Better quality: Local models can concentrate their full expressiveness on their dedicated sub-problem space. Representing important properties of certain queries in the training data to predict the best hint set is easier because the local model can focus on these properties. The separate context models can distinguish fine-grained hint combinations per query that would otherwise be overshadowed by queries of completely different contexts. Therefore, the local approach outperforms the global models in quality because its models can focus on particular properties to find the best hint set. Even very detailed changes in the predicates of a query can be picked up on and used for stable predictions. The dynamic model placement also covers the changes in joined tables defining the contexts, creating contexts according to the training workload and new unknown queries.

(A3) Faster training: The smaller models of the local approach can reach better qualities in predicting hint sets than global models. However, they are also faster in training. On a per-context level, they require far fewer queries to reach high-quality hint set predictions than global models. Following the higher sample efficiency, there is a reduced need for training queries. This makes the training and labeling process faster. We also introduce a modified labeling strategy that further improves the training performance.

(A4) Faster forward passes: Again, we gain performance in the model application or forward pass with the smaller and less complex models. The fewer calculations per model and per context make the direct prediction of hint sets efficient. This enables the optimizer to use the local models during runtime while adding only negligible overhead to query optimization. This is important because hinting has to be done query-wise in a running DBMS. Therefore, we expect many forward passes, i.e., one per incoming query, for the local models.

4.4 SMART LABELING

Supervised learning requires a labeled set of training queries where a label contains the best, i.e., optimal, hint set for the query. Retrieving these labels, or *labeling*, is a time-consuming step, but it only needs to be run once for a fixed set of training queries. Our labeling needs to retrieve the optimal runtimes and the corresponding hint sets for all queries in the training data. The naïve approach runs every query using every possible hint set, stores the corresponding runtimes and hint set, and chooses the best ones for every query. This is very ineffective as it is not only very time-consuming but also stresses the database enormously because we would execute a lot of queries at once.

To overcome this issue, we define a *smart labeling* procedure. The general idea is the introduction of a runtime timeout per query, after which the query is aborted and the corresponding hint set can be safely discarded because it is not the best one. Our labeling algorithm iterates over all queries in a training data set, which is a subset of a given workload. Initially, the current query is executed with the PostgreSQL default hint set and the resulting response time is set as a timeout as we are only interested in hint sets with a reduced response time compared to the default runtime. To further speed up the labeling process, we aggressively lower the timeout whilst evaluating each query under every hint set. Whenever a hint set leads to a response time lower than the current timeout, this new response time replaces the timeout. Additionally, the best hint set for every query is stored. Thus, the labeling process speeds up over time when better hint sets with lower query response times are retrieved for a query. This leads to reduced runtimes even for larger training data sets while generating the complete set of optimal hint set labels.

4.5 RUNTIME MODEL MAINTENANCE

The supervised approach of the local learning strategy for query optimizer hinting might be too rigid since every model within each context is fixed after training. Thus, it is likely that the local strategy might not predict an ideal hint, e.g., due to data or workload shifts. To overcome this issue, a naïve approach would be to initiate a retraining with every incoming query. This would be possible from a model point of view because each query is assigned to exactly one context model and only this model has to be retrained. In our case, the retraining of a single GB model is negligible. However, any frequency of retraining during runtime already makes this approach impractical. This is further complicated by the fact that for each retraining, the corresponding triggering query would have to be executed with all 2^k possible hint sets to determine the optimal hint set for the retraining. Although this labeling can be optimized as described in Section 4.4, this still creates a high load on the DBMS.

Since we cannot completely avoid retraining to achieve good model qualities all time, we propose a novel approach. Instead of triggering a retraining with every incoming query, we want to actively initiate a retraining of a context model when an update is necessary. To recognize fitting situations, we take advantage of the divide-and-conquer properties of our approach. Once incoming queries are executed with a predicted hint set, we also supply a per-context timeout to the query compiler such that abnormally long-running queries are detected. These long-running queries indicate that the model needs to be refreshed. The timeout is calculated for each context model based on the best response times for every query collected during labeling. Timeout t is then a mapping for every context c, corresponding observed best times T_c for all context queries during labeling, and a combination of absolute and percentage-based relative timeout $t_a \in \mathbb{R}^+, t_p \in \{0\%, \dots, 100\%\}$ as follows:

$$t(c) = \max(t_a, percentile(T_c, t_p))$$

This runtime maintenance has a variety of implications for our model. Firstly, the percentile-based timeouts and, therefore, choosing a value in the range of observed execution times allows us to enforce different retraining behaviors. A small value t_p enforces greater retraining rates as the probability of exceeding a smaller timeout increases. Contrarily, high values of t_p may decrease the retraining rate. Moreover, using an absolute threshold t_a implies a minimal timeout value. This is especially useful for contexts consisting solely of fast-running queries with lower query response times than t_a. Hinted queries will then not trigger a timeout until exceeding t_a. This value can be chosen in a workload-dependent manner. Notably, the chosen per-context timeouts are not static throughout the strategy's lifetime. Upon retraining within a context, abnormal queries are asynchronously labeled and then added to our training data. After each retraining, we add the optimal hint set time of the labeled queries to our experienced times T_c. The corresponding timeout is then updated accordingly. This retraining introduces runtime adaptivity to our supervised classification approach, making it more flexible and precise during application runtime.

4.6 EVALUATION

In this evaluation, we show the advantages **(A1)** to **(A4)** of our local strategy for query optimizer hinting. We detail the influence on workload performance with several experiments regarding end-to-end runtimes. Furthermore, we show the influence of the amount of training data and the contextualization of the problem spaces on the local models' quality. Lastly, the impact of the runtime model maintenance is evaluated as to how its effecting workload *speedups*.

Table 4.1: End-to-end runtimes and speedups (in brackets) of our local learning strategy for different benchmarks.

Benchmark	Configuration	PG 12.4	PG 14.6
Stack	PG default	19,384s (1.0x)	9,087s (1.0x)
	Optimal hint set	7,234s (2.6x)	3,561s (2.5x)
	BAO	14,666s (1.3x)	n/a
	Local strategy	7,760s (**2.5x**)	3,649s (**2.49x**)
JOB	PG default	204s (1.0x)	185s (1.0x)
	Optimal hint set	87s (2.3x)	79s (2.3x)
	BAO	243s (0.85x)	n/a
	Local strategy	88s (**2.3x**)	95s (**2.0x**)
TPC-H	PG default	445s (1.0x)	143s (1.0x)
	Optimal hint set	119s (3.7x)	110 (1.3x)
	BAO	388s (1.15x)	n/a
	Local strategy	137s (**3.25x**)	127s (**1.15x**)

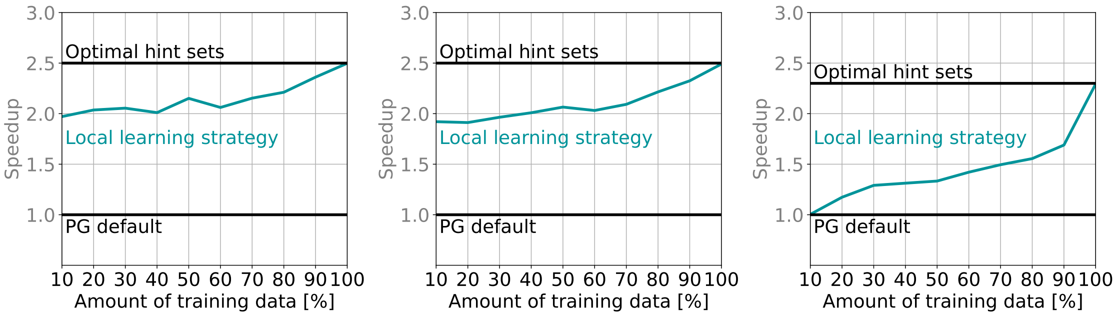
4.6.1 Setup

To evaluate the benefits of our local learning approach, we use three different benchmarks: Stack [54], Join-Order-Benchmark (JOB) [49], and TPC-H [22]. The Stack benchmark consists of ten relational tables and 6,191 workload queries, which can be divided into eleven different contexts. Each context joins a different number of tables, ranging from three to eight. While seven contexts contain up to 200 queries, the remaining four contexts consist of more than 1,000 queries each, the largest having circa 2,000 queries. The JOB workload has a total of 113 queries distributed over 33 contexts, while TPC-H has 22 workload queries over 18 contexts. All workloads contain different kinds of analytical queries, including SPJ queries and aggregates.

All experiments are executed on an Intel Xeon Gold 6216 system with 92 GB of main memory and 1.8 TB of HDD storage. To test the performance of our approach on different systems, we use the PostgreSQL (PG) versions PG 12.4 and PG 14.6 as underlying DBMSs. They represent two different systems because their inner workings are dissimilar due to major release changes. However, the optimizers of both systems can be steered with the same six Boolean hints, as described in Section 4.3. Each experiment is run several times, each run with a different random query ordering. Therefore, the following results are consistently averaged values over five runs.

4.6.2 Benchmark Performance

To evaluate the end-to-end performance of our local learning strategy for query optimizer hinting, we run all queries of all three benchmarks, both with the PG default hint set and the optimal hint set on both PG 12.4 and PG 14.6. The *optimal hint set* produces the lowest query runtime out of all possible hint sets and is retrieved for each query individually. Additionally, we let our local strategy select the best hint set according to its predictions and execute the query with it. Accordingly, we ask BAO for its best hint set and execute the query. For all experiments, the runtime model maintenance from Section 4.5 is enabled. This is done for the whole benchmarks on the two systems. However, BAO only works on PG 12.4 for compatibility reasons and its deep integration into this



(a) Speedup Stack on PG 12.4. (b) Speedup Stack on PG 14.6. (c) Speedup JOB on PG 14.6.

Figure 4.5: The local selection strategy for query optimizer hinting for different workloads and systems.

version of PG. All runtimes and speedups for the four approaches, three benchmarks, and two systems are detailed in Table 4.1. The learned models were trained with *train-to-represent* where all, i.e., 100%, workload queries are used in training. It is visible that our approach outperforms the other two approaches, i.e., PG default and BAO. Additionally, it shows performance close to the optimal hint set performance. Therefore, the local learning strategy is able to predict the optimal hint set for most queries in a workload. It can accelerate workloads up to factor 3.25x by just steering the optimizer through its hint but without interacting with the optimizer directly. Through the comparison with BAO and the better workload performance, advantage **(A2)** better quality can be demonstrated for the local learning strategy.

This experiment also implicitly details advantage **(A4)** faster forward passes. Although the forward passes are not listed explicitly, the faster workload runtimes argue in favor of the local learning strategy. The low overhead of our strategy enables more speedup per workload because the decision-making for predicting a hint set per query is faster than for global models like BAO. Therefore, the faster forward passes facilitate the local learning strategy to reach near-optimal runtimes according to the optimal hint set.

4.6.3 Amount of Training Data

Another important part of our local learning strategy is stability and generalization. Usually, we cannot use all workload queries for training learned models but just a subset of them because, during runtime, a workload is mostly unknown to the DBMS. The local learning strategy needs to generalize from little training data to the whole workload. To show the quality of our local learning strategy with less training data, we set up an experiment where we increased the amount of training from 10% to 100% in 10% steps. The queries that are not in training data are automatically used for testing. For example, a *20-80 split* or just *20% split* means that 20% of the workload queries are used for training and 80% for testing by running them under predicted hint sets. Therefore, the test queries are completely unknown to the local models, which have to generalize from the training data to predict the test query hint sets. The rest of the experiment is designed in the same way as in the previous section. Figure 4.5 details the speedups for the test queries as a teal line with three scenarios: Figure 4.5a for the Stack workload on PG 12.4, Figure 4.5b for the Stack workload on PG 14.6, and Figure 4.5c for the JOB workload on PG 14.6. Therefore, we are able to show both the benefits of the local learning strategy for different workloads and systems. For all workloads and systems, the local learning strategy for query compiler hinting (i) takes little training data to reach high speedups

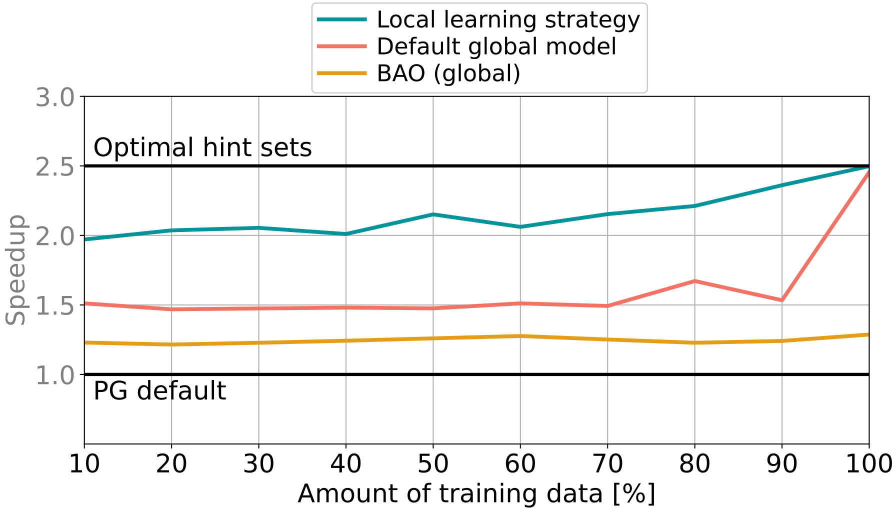


Figure 4.6: Speedup of local and global models for Stack on PG 12.4.

and (ii) approaches the optimal hint set performance if more training data is used. For JOB, the range of reached speedup is wider because it only contains 33 queries. So, for a 10% split, only three queries are used for training, which might not be enough for the models to generalize to the whole workload. However, the speedup also increases with the increasing number of training queries.

Therefore, we demonstrate that advantage **(A3)** faster training is also fulfilled because the local learning strategy for query optimizer hinting reaches high speedups with less training data. As for the cardinality estimation models, the training time of the local models for hinting is linearly reduced if less training data is used. The training time can be reduced without sacrificing quality. This also allows for the rapid creation of new contexts if unknown queries, i.e., from unknown contexts, are introduced. Further experiments show that we need as little as two queries per new context to generate speedups higher than one [95].

4.6.4 Contextualization

To test the influence of contexts on the local models' quality and to compare it to global models without contexts, we derive another experiment. In Figure 4.6, we use the same training-test splits as in the previous experiments and add two more global models to be trained on these splits. Firstly, we switch off the context-aware part of the local learning strategy. This builds one large GB model for all queries in the Stack workload simulating a *default global model* with no contextualization. Secondly, we also compare our approach to BAO as a global model contender. Again, BAO only works on PG 12.4 because of its deep integration within the system. Therefore, the experiment is conducted in PG 12.4. We also include the PG performance under the default hint set. Our local strategy's performance, i.e., reached speedups, is better than the performance of the two global models. From the difference between the local approach and the default global model, we can see that contextualization helps our local learning strategy to break down the complexity of the whole problem space. Introducing contexts produces better query runtimes and higher speedups through better hint set predictions than a one-size-fits-all solution. Additionally, the speedup gain is higher from split to split for the local learning strategy, meaning it can generalize better with less training data, as shown in the previous section. The same argument holds for BAO. Furthermore, our local learning

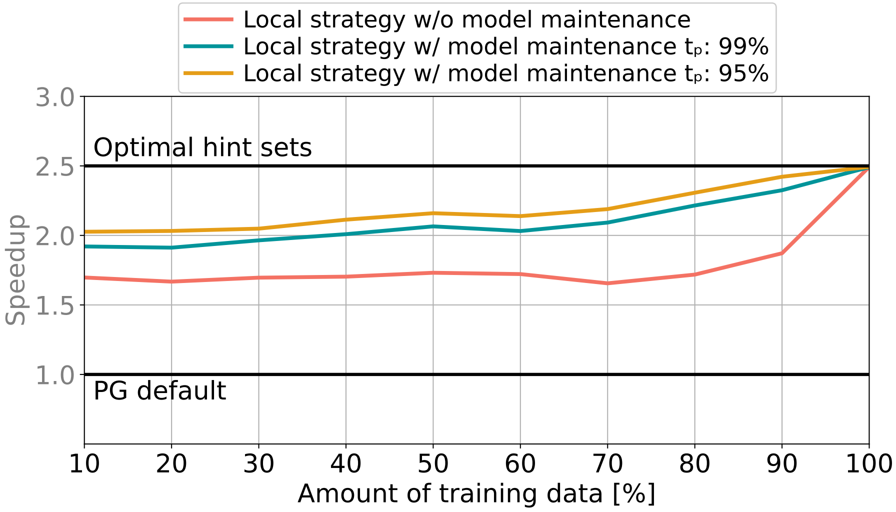


Figure 4.7: Speedup runtime model maintenance for Stack on PG 14.6.

strategy is better than BAO because BAO does not reach its promised potential. BAO does not reach the optimal hint set performance with train-to-represent, unlike the other two approaches. This is because BAO relies on the underlying system configuration and cannot generalize its predictions if necessary [95]. Additionally, the implementation of BAO only uses five different hint sets due to performance limitations [54], whereas our local learning strategy utilizes all 64 hint sets.

We can demonstrate that through contextualization advantage **(A1)** less complex models is accurate because the local learning strategy for query optimizer hinting reaches higher speedups with contexts and the same amount of training data than global approaches without contexts. The complexity of the local learning strategy is less because the combination of higher quality and less training data requires less complex local models and sub-problems in every context. Otherwise, if the local models and sub-problems were complex, one of these two attributes would not be better than for global models. Additionally, we rely on GB models and their lower complexity, as shown extensively in the previous chapter. We will also extend the analysis of GB models in the next chapter.

4.6.5 Runtime Model Maintenance

A specialty of our approach is its adaptability during application through the runtime model maintenance. Unlike the reinforcement character of BAO, our supervised local learning strategy needs additional planning to adapt to changes during runtime. To evaluate our runtime model maintenance, we run the experiment for Stack on PG 14.6 from Figure 4.5b without runtime model maintenance. Furthermore, the plot in Figure 4.7 contains speedups for runs with enabled runtime model maintenance with different relative timeout thresholds $t_p \in \{95\%, 99\%\}$ for the dynamic timeout calculation over the already measured query runtimes. We derive three conclusions from Figure 4.7. Firstly, activating the runtime model maintenance improves the speedup of the Stack workload. Secondly, the overhead of the runtime model maintenance, including the cancellation of queries, the rerun of the queries with PG, and the retraining of the local models are low enough to still generate speedups. Thirdly, a lower threshold for the relative timeout leads to better speedups and earlier speedup gain. With a lower threshold, queries are marked earlier to be aborted and used for retraining. This leads to better predictions by the local models for these complex, long-running queries. Therefore, the local learning strategy is able to adapt to complicated queries and changing workloads. The runtime model maintenance is a beneficial component of our local learning strategy for query optimizer hinting.

4.7 SUMMARY

In this chapter, we introduced and applied a local learning strategy for query optimizer hinting. Through the detailed evaluation, we can demonstrate that the local learning strategy for query optimizer hinting fulfills the four advantages **(A1)** to **(A4)**.

(A1) Less complex local by dividing the problem space into sub-problems, i.e., the predicates in a set of joined tables

(A2) Better workload runtimes and speedups through a direct hint set prediction per query, specialized models per context, and model maintenance during runtime

(A3) Faster training times by using less training data per local model while reaching better qualities and the fast deployment of models in unseen contexts

(A4) Faster forward passes from less complex local models

The advantages of our local learning strategy were evaluated by comparing its performance to traditional and learned global approaches. We have shown the different components of the learning strategy that are important for query optimizer hinting and their influence on end-to-end performance. Our learning strategy works with different workloads with different properties. We compared three different standard benchmarks and could show that we reached speedups up to 3.25x compared to default optimizer execution. Additionally, the local learning strategy for query optimizer hinting can be applied to different DBMSs. With a common interface, like SQL, we can steer the optimizer of any DBMS by setting hints and hint sets on a per-query level. The four general advantages of the local models, the specific qualities of our approach for optimizer hinting, and the system independence make our local learning strategy for query optimizer hinting a valuable addition to query optimization and processing. Furthermore, through smart labeling and runtime model maintenance features, the local learning strategy for query optimizer hinting features techniques to be applicable to changes in workloads and data. Overall, it improves the quality and performance of the query optimizer without direct interference or integration.

5

THE LOCAL LEARNING STRATEGY FOR INTEGER COMPRESSION SELECTION

Integer compression has been a well-established query optimization component in DBMS for decades [21, 33, 75], especially for in-memory column stores [2, 3, 26] to optimize the execution of analytical queries. For column stores, lightweight integer compression algorithms play an essential role since all columns are usually encoded as sequences of integer values. Based on that encoding, the whole query processing is done on these integer sequences [2, 3, 12, 29, 83]. On the one hand, integer compression can reduce the necessary memory [2, 3, 24]. On the other hand, compressed integer values allow for improving the processing performance (i) by increasing the effective bandwidth to reduce the memory wall, (ii) by yielding a better utilization of the cache hierarchy, and (iii) by enabling highly data-parallel processing [2, 3, 24, 26]. However, choosing the right compression algorithm is not trivial. The different objectives of query optimization lead to high complexity in selecting the right algorithm for a given data set. Most algorithms only achieve improvements for one objective at a time, so a holistic selection has to look at all combinations of algorithms, data properties, and objectives to reach improved results.

For example, the column store MorphStore [26] uses an operator-at-a-time processing model with on-the-fly (de)recompression, i.e., operators decompress the inputs and recompress the outputs on the fly while processing uncompressed data internally. Thus, the (de)compression runtimes determine the query runtime difference between two alternative query plans with different compressed formats selected. In column stores, it is crucial for query performance to choose the right algorithm or an algorithm close to the performance objective optimum. Therefore, a key feature of a selection strategy is the inclusion of general knowledge about data distributions and their influence on the algorithms to make a data-independent decision and generalize over different inputs.

5.1 PRELIMINARIES FOR LEARNED COMPRESSION SELECTION

To assess the full capabilities of integer compression and their influence on query performance, we collate five objectives.

O1: compression rate,
O2: runtime compression (ram2ram),
O3: runtime decompression (ram2ram),
O4: runtime decompression (ram2reg),
O5: runtime compression (cache2ram).

This gives us a broad overview of the different targets a selection strategy should fulfill. The compression rate shows how much memory can be saved when using a particular algorithm. The following four objectives detail the times for compressing and decompressing the data. These are split into the compression mode from memory to memory (ram2ram) and cache to memory (ram2reg, cache2ram). Objective O4 is called ram-to-register because decompression is not done into cache lines but directly into registers [24]. All four runtime objectives are important measures for query processing. With these five objectives, we are now able to differentiate the advantages of related and our work regarding their impact on integer compression selection.

Again, we will focus on the four shortcomings of other approaches using global models in Section 5.2. To achieve a (user-)data-independent selection approach for all objectives, we propose a *local learning selection strategy for integer compression selection* and its advantages in Section 5.3. For this and to eliminate the manual effort, human engineering,

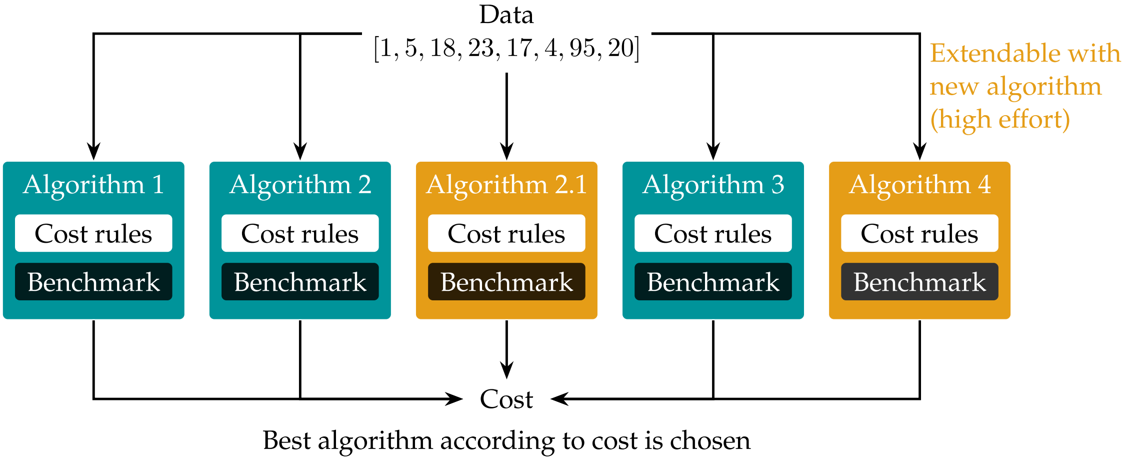


Figure 5.1: The cost model for compression algorithm selection by Damme et al. [25].

and data dependency, our local learning strategy views compression algorithms as black boxes, i.e., local ML models. The local learning selection is split into three core components: data generation, feature engineering, and models. The data generation and feature engineering processes are presented in Sections 5.4 and 5.5. Additionally, the divide-and-conquer characteristic of our approach allows us to use lightweight models, which will be discussed in Section 5.6. The black-box approach ensures broad applicability and generalization over different data to model the required knowledge for decision-making. The local learning-based approach enables us to use the advantage of training models once on a synthetic data set from the data generation component and then apply them repeatedly to new unknown data with very little overhead through their fast forward passes. We evaluate all these properties in Section 5.7 before summarizing the advantages of a local learning strategy for integer compression in Section 5.8.

5.2 RELATED WORK

The large number of proposed integer compression algorithms in the literature and the insight that there is no single-best algorithm highlight the complexity of finding the best algorithm for every data input [24, 25, 51]. The best integer compression algorithm depends on data as well as hardware properties. To select the best-fitting compression algorithm, an appropriate selection strategy is needed. For that, three general approaches have been proposed: (1) rule-based selection [3], (2) cost-based selection [25], and (3) ML-based selection [11, 15, 40, 41].

5.2.1 Rule-based Selection

Regarding rule-based techniques, Abadi et al. [3] have hand-crafted a decision tree based on an empirical evaluation of a small number of compression algorithms. One advantage of this approach is that making a decision is very cheap as it requires only a few steps through the tree. However, as the field of lightweight compression has evolved significantly since then, their decision tree does not cover (i) the diversity of the algorithm landscape nowadays and (ii) the hardware dependency.

5.2.2 Cost-based Selection

In the area of cost-based selection, Damme et al. [25] envision a cost model for lightweight integer compression algorithms. This cost model adopts a *grey-box approach* by explicitly modeling as much knowledge about the algorithms as possible (white box) and implicitly capturing the impact of data and hardware characteristics using a small number of calibration benchmarks for each algorithm (black box). The cost model is depicted in Figure 5.1. Each of the five algorithms has two parts: the hard-coded cost rules as explicit knowledge and the benchmark statistics as black box knowledge. The cost model can derive the cost of a model under a specific objective by combining the information from both boxes. The algorithm with the lowest costs is chosen as the best one. While this approach is able to generalize to a wide range of lightweight integer compression algorithms, a certain amount of manual effort is required to include a new algorithm. This includes the induction into the algorithm's definition, the iterative implementation of the algorithm, and different benchmarks for the algorithm. Furthermore, making a single decision with the cost model involves much more calculations than traversing a decision tree. Given the multitude of available compression algorithms, scoring each to select the best one can become very expensive.

5.2.3 ML-based Selection

A major shortcoming of both traditional approaches is the manual effort required to incorporate algorithms into the selection process. So, a third category of selection strategies based on ML has been proposed to avoid this effort. ML-based selection strategies are a relatively new contribution to the field of compression [11, 15, 40, 41]. They model the general selection process as a black box. Data properties are put into a learned model and the potentially best algorithm is predicted by the model as a classification task. These models produce good-quality decisions fast. Additionally, they are trained on one data set and tailored to work on that particular use case and objective. This reduces the generalization capabilities of the ML models and makes them data-dependent.

Jin et al. [41] model the selection problem as a classification task. During training, they exhaustively determine the best algorithm for a set of training data blocks. Then a test data block is assigned to the training data block with the most similar data properties and the known best algorithm for that training block is selected.

Another example of ML-supported selection is CodecDB [40]. CodecDB uses a learned global model classification approach to find the best algorithm according to the compression rate objective. However, this is the only objective that this model considers. Other objectives that would be crucial for query performance, like O2 to O5, are not modeled in the original paper, but for our evaluation, we extended the model to be usable with all objectives. The model is a neural network with 14 neurons in one hidden layer and hyperparameters as described in [40].

Furthermore, LEA [15] is an ML-based compression selection approach for column stores. LEA trains models to predict the best column data representation for optimized query execution. LEA uses synthetic data for training. However, like CodecDB, LEA currently only focuses on the data compression rate and also relies on cardinality estimates, which can negatively influence the optimizer due to their high estimation errors, as shown in Chapter 3. Additionally, it uses sampling from the input data during the forward pass, which might slow down its application during runtime or obfuscate important data properties.

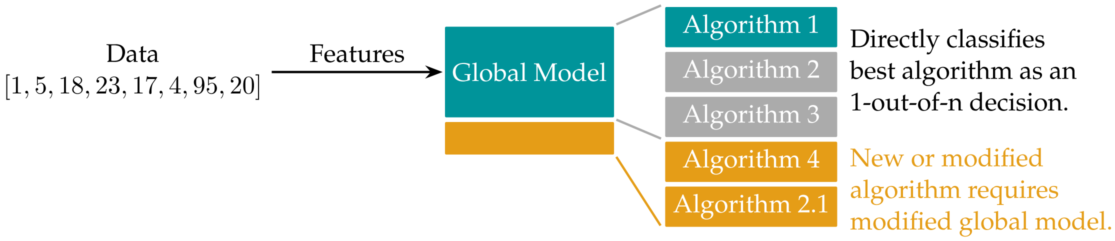

Data
[1, 5, 18, 23, 17, 4, 95, 20]
Features
Global Model
Algorithm 1
Algorithm 2
Algorithm 3
Algorithm 4
Algorithm 2.1
Directly classifies best algorithm as an 1-out-of-n decision.
New or modified algorithm requires modified global model.

Figure 5.2: The global learning selection strategy as a classification problem.

5.2.4 Shortcomings of Global Strategies

All three ML-based approaches are based on a global model classification problem as depicted in Figure 5.2. The global model directly predicts the best algorithm from data-derived features. This is done by making a one-out-of-n decision over all possible algorithms. While this works, it discloses the same shortcomings as the global models for cardinality estimation and query optimizer hinting.

(S1) Large complex models: Due to the one-out-of-n decision of the classification approaches, a lot of intelligence is needed to make an informed decision. Global selection strategies models are getting complex fast because of these prerequisites. A lot of predictive power is required by the ML models and to match this larger models are designed. Therefore, just like for cardinality estimation and query optimizer hinting, global models are larger in memory footprint, harder to maintain, and more complicated to use.

(S2) Overfitting: Additionally, through their large and specialized architecture, global models tend to overfit. This can be due to non-representative training where one part of the problem, i.e., an algorithm or objective, is misrepresented in the data, feature, or even problem description. The global models mirror this representation to its predictions lowering the overall quality of the selection process by choosing wrong or suboptimal algorithms. A global model is also prone to data shifts because it cannot react to them without complete retraining. The same is true for changing problem spaces. In integer compression selection the underlying collection of algorithms is changed quite often, most of the time because new algorithms are added or existing algorithms are modified. A global model has to be retrained from scratch to be able to cover the changes.

(S3) Amount of training data: Another general disadvantage of large complex models is the amount of training data. The larger the model and the more complex the architecture, the more training data is required to formulate an expressive model. This automatically increases training times and makes the model lumbering regarding different types of adaptivity.

(S4) Slow forward passes: For a runtime decision, integer compression selection strategies need to be fast during application. When applied during query optimization, the forward passes through the model need to be fast. However, global models, because of their complex structure, have slow forward passes. This is due to the fact that higher complexity dictates more calculations for a single forward pass.

In the next section, we detail our *local selection strategy for integer compression* and how it is capable to support the qualitative selection of integer compression algorithms for query processing, especially in column stores. Additionally, we derive its advantages according to the shortcomings **(S1)** to **(S4)**.

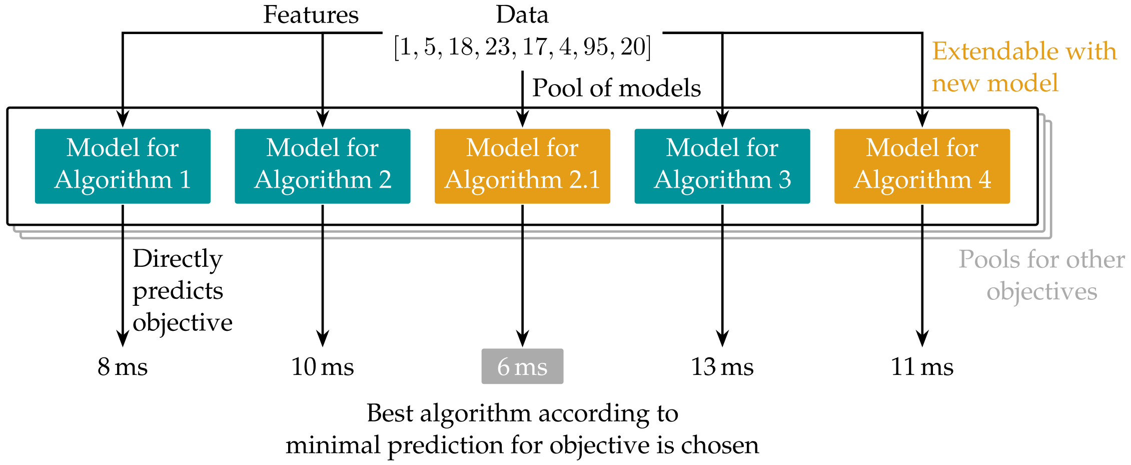


Figure 5.3: The local learning selection strategy with regression models for one objective.

5.3 THE LOCAL LEARNING STRATEGY FOR INTEGER COMPRESSION SELECTION

Similar to the presentation of local models for cardinality estimation and query optimizer hinting, we transfer the application of a local divide-and-conquer approach to compression selection. Figure 5.3 shows our *local selection strategy for integer compression* based on regression for a single objective as first presented in [89]. The results and parts of this chapter are based on this publication. In contrast to related work based on classification, as presented in Figure 5.2, the local strategy splits the decision for the right algorithm into sub-problems. Therefore, it directly manifests the divide-and-conquer technique representative of a local learning strategy. For this, we split the problem space into sub-problems with a single model for each algorithm and objective combination. Therefore, a *context* for integer compression selection is defined by an algorithm-objective combination. A single local model in such a context is seen as a representative of the algorithm under the objective. The model is then trained on example data to predict the objective as a regression model target for a particular algorithm. For our example in Figure 5.3, there are five models and the five objectives from the beginning of this chapter. In total, we generate $5 \cdot 5 = 25$ models in our *pool of models*. For better illustration, Figure 5.3 only shows the pool for one objective. Anytime a new array or column of data arrives, the features of the data block will be forwarded to all models of a specific objective. Each of these models then directly predicts the objective on the data under the algorithm that the model is representing. Given all predictions, the local selection strategy selects the model with the minimal prediction and returns the corresponding algorithm as the best algorithm to the user. It is important to highlight that during this process, no compression algorithm has to be executed.

Advantages of the Local Learning Strategy

From the structure of our local compression selection for integer compression, we can directly derive four advantages circumventing the four shortcomings of global strategies.

(A1) Less complex models: The division of the problem leads to smaller models for each sub-problem. These models are less complex than the global models because they do not need to model the one-out-of-n classification as a dependency on other algorithms like global approaches do. The performance of a compression algorithm does not depend on any other algorithm's properties or performance. The only interdependent part of two algorithms is which one is better according to the objective. Our local approach takes this prerequisite and models each compression algorithm independently. Additionally, we can show that the total size in memory of the pool of models is smaller than a global model architecture.

(A2) Better quality: With the smaller, focused models, we are able to concentrate more expressiveness of the models on the sub-problems. This automatically leads to better prediction qualities for the directly predicted objective per model and the overall prediction by the pool of models because the model can better concentrate on the properties and specialties of a single compression algorithm. Additionally, the focus on single models allows for easy expandability if new algorithms are introduced or existing algorithms are modified. Therefore, the local selection strategy can outperform related approaches in quality and solves one of the major drawbacks of global ML models. We detail these improvements in the evaluation.

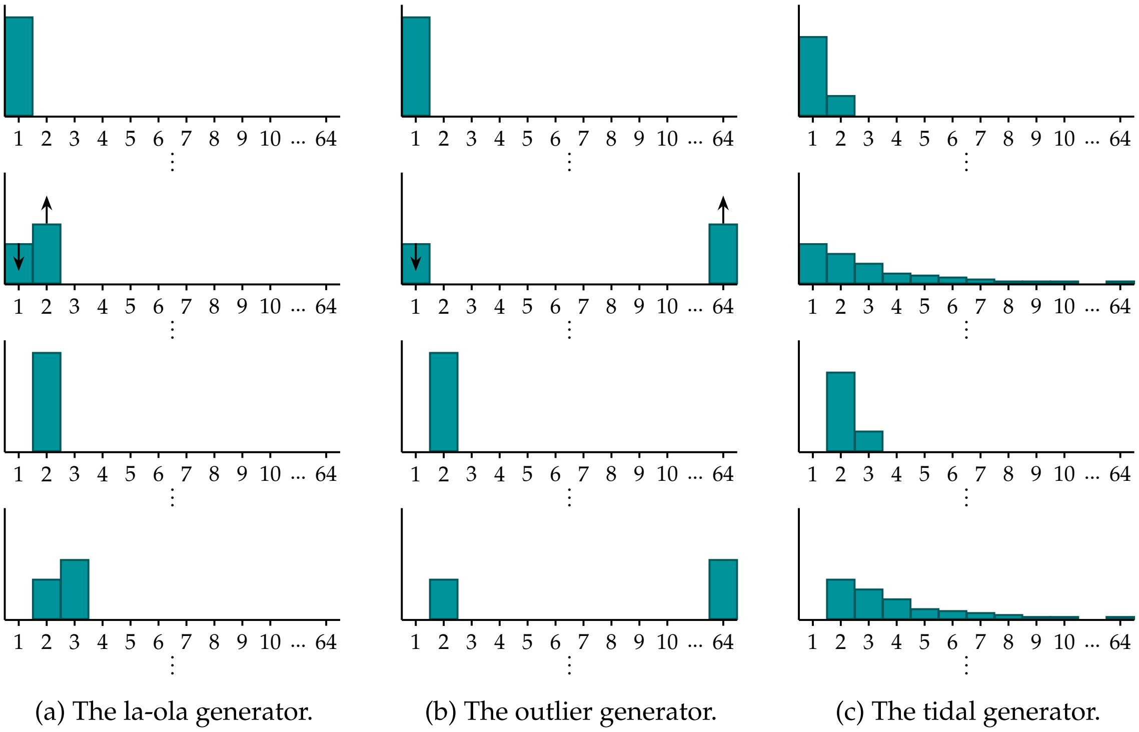
(A3) Faster training: Through the lesser model complexity for local models, the training is faster compared to global models. This is due to two factors. The first one is that a smaller model is faster in training because it has fewer parameters to learn. This intrinsic characteristic is true for any ML model [10]. The second point is the lesser required training data to reach an equivalent level of quality for local models compared to global models. This point follows the same argumentation as the local selection strategy for cardinality estimation in Section 3.3 and query optimizer hinting in Section 4.3. The smaller models can train more expressiveness with fewer example data.

(A4) Faster forward passes: With the smaller model structure, a single model application, i.e., retrieving the objective for a given data array, is also faster. This is important for our local strategy because we need to query several models at once to find the minimal prediction over all algorithms. Here, the faster forward passes help us to retrieve faster predictions over several models from the whole pool than global models. In our evaluation, we will show that the prediction time for one objective is faster with our pool of models than for comparable global models.

To reach the four advantages **(A1)** to **(A4)**, we need to focus on three parts in setting up our selection strategy. The following sections examine three core components and describe their functionality. Firstly, Section 5.4 explains a *user-data-independent* data generation allowing for the training of models even though there is no (user) data. Secondly, we need to select meaningful features to be derived from the data and forwarded to the models in Section 5.5. Lastly, we have to consider the best-fitting model type for the pool in Section 5.6.

5.4 DATA GENERATION

In general, training supervised regression models needs example or training data. However, most of the time, no data is available when setting up the pool of models within a system. This is referred to as the *cold start problem* [89]. To avoid this problem and to be able to start training our models right from the start, we propose a data generation that produces artificial user-data-independent training data for our models. This data generation is required to cover the whole problem space P_l of a local sub-problem, i.e., the stable prediction for a single algorithm and a specific objective. Additionally, the generated data points have to be executed once on the target hardware to retrieve the regression labels. In our case, this is the hardware on which the selection is used during



(a) The la-ola generator. (b) The outlier generator. (c) The tidal generator.

Figure 5.4: Different user-data-independent data generator techniques for BWHs.

application. The labels are the measurement results of all objectives on the target hardware for the artificial data. Both the data and the labels fulfill the minimum requirement to train our local models.

To cover the problem space in the best possible way and generate meaningful representative data, we need to generate as much data as possible, but only as much as is required. To keep this balance, we base our data generation on *bit width histograms* (BWHs) that count the occurrence of integers per bucket. Each bucket has an index b telling that this bucket collects all integers using the b^{th} as the highest bit set to one. BWH buckets can either use the absolute count or relative percentage of integers per bucket. Figure 5.4 exemplary shows several BWHs. The most important argument for BWHs is that we directly derive features and example data for label generation from them. Additionally, generating every possible integer array with variable length to cover the whole problem space is not feasible. BWHs reduce this sampling effort by limiting the sample space to every possible bit width distribution. However, this would still be too much effort because a relative BWH for 64-bit integers with a maximum of 100% in 1% steps per bucket would still lead to 100^{64} different BWHs. So, in our three generators, we reduce the parameters of a BWH by fixing the step size to 2% and only using important distributions according to the generator's idea to model the problem space. Therefore, we define three generators with three different approaches to generating BWHs: the *la-ola generator*, the *outlier generator*, and the *tidal wave generator*. Each generator follows a different idea to model the integer data problem space. All generators are pictured in Figure 5.4.

La-ola generator: Integer compression algorithms are very sensitive to the largest bit width used in the data [24, 25]. So, for our first generator, we incorporate this idea by using two (and only two) neighboring buckets for the two highest bit widths. We start with a completely filled bucket and begin to move 2% to the next lower neighboring bucket until this one is filled. Then, we restart the process with the now-filled bucket as the next starting point. This is repeated until all neighboring bucket pairs underwent this process.

Table 5.1: Important features for integer compression data.

Feature name	Feature description	Feature group
sorted	Is the data sorted?	sorting
minBucket	Minimal bucket occupied in BWH	BWH properties
maxBucket	Maximal bucket occupied in BWH	BWH properties
min	Minimal % over all buckets in BWH	BWH properties
max	Maximal % over all buckets in BWH	BWH properties
numBuckets	Number of filled buckets in BWH	BWH properties
avg	Average bit width of the data	data statistics
std	Standard deviation of the data	data statistics
skew	Skew of the data	data statistics
kurt	Kurtosis of the data	data statistics

Every repetition generates a new BWH. Figure 5.4a shows some of the generation steps and resulting BWHs. The generated BWHs form the titular la-ola wave, sometimes also called a stadium wave.

Outlier generator: Another influence on integer compression algorithms is outliers in the data [24, 25]. Here, an outlier is an integer with a different bit width than the other data. To model outliers in BWHs, we modify the la-ola generator to use two buckets with the most distance between each other instead of neighboring ones. The rest of the generation process is kept the same. This leads to the BWHs as shown in Figure 5.4b.

Tidal wave generator: The last generator, the tidal wave generator, picks up the idea that integer compression algorithms depend on the general data distribution over all buckets of a BWH [24, 25]. Therefore, we use different bit width distributions in the BWHs by generating the BWH directly from Zipf distributions. This can be seen in Figure 5.4c. The final BWHs look like a tidal wave coming ashore.

With any of these three generators, we generate enough data to produce high-quality models and also minimize the time spent on label retrieval. We will show the different qualities and performance of each generator in our evaluation and derive general recommendations on when to use which generator.

5.5 FEATURE ENGINEERING

Another important aspect of our local selection strategy is feature engineering. *Feature engineering* is the use of condensed information from data to deliver meaningful information to a model [76]. In our case, we need to find the most important data properties that can be derived from a bit width histogram (BWH) to keep in line with the ideas of our data generation from the previous section. Additionally, we need to tackle the problem of integer arrays with different lengths.

Firstly, we solve the problem of different lengths of input data arrays by using *blocking*. Each input array or column is split into sub-parts, called *blocks*, with a fixed number of integer values. The number of values per block is flexible but needs to be fixed for a single local selection strategy pool. Then, for every block, a BWH can be created. From this BWH, the features are derived. The blocking leads to fixed-length inputs for the BWH and, therefore, for our models. Every block is now handled like a whole column

Table 5.2: Performance of neural networks and gradient boosting.

Model	Training time [s]	Hyperparameter tuning [s]	Forward pass [µs]
GB	0.12	2.4	0.8 (±0.2)
CodeDB	13.20	-	56.0 (±1.3)

by the pool of models. This allows for a blockwise processing of columns in a column store, meaning we can use a different compression algorithm on sub-parts of a column according to their properties during runtime. In our generation approach, we generate blocks of integer data instead of columns from the BWHs to keep congruent with the idea of blocking. Therefore, we can generate blocked data for both training (from our generators) and application (from real-world data).

Damme et al. [24, 25] collect several ideas for important features based on integer compression algorithms' properties. They identify three groups of features: sorting, BWH properties, and data statistics. We list all features identified by related work in Table 5.1. Except for the first group, all features can be derived from BWHs efficiently. All ten features are seen as equally important for the behavior of integer compression algorithms [24, 25]. Throughout this chapter, we keep this set of features fixed for every aspect of our local selection strategy.

5.6 MODELS

The last important piece for our local learned selection strategy is the type of models in the pool. The model type is key to building a fast selection strategy in training and forward passes. To keep these two parts as short as possible, we decide not to use neural networks (NNs) because they would induce long training times. Additionally, we want to generate as little example data as possible and NNs perform best with lots of training data. Given these prerequisites, we decided to use Gradient Boosting (GB) for local models, as presented in Section 3.4. This technique has the advantage of being lightweight because its modeling is based on weak predictors, i.e., decision trees that have a fast training phase and forward pass. In Table 5.2, we compared the performance of a single local GB model with CodecDB [40], a competing global model based on a NN. Both models were trained on a training data set containing features from 3,151 BWHs. The GB models are factor 100 faster in training and factor 70 faster in forward passes. Therefore, we argue that GB should be used preferably for local models in the selection strategy pool.

Again, these GB models have two hyperparameters that mainly influence model quality: the number of estimators P and the maximum depth per tree estimator d. Any change in these has an impact on the model's final quality. As a result, we need to find the best combination of P and d. The *hyperparameter tuning* is done by repeating the training of the GB model with different parameter combinations. A test data set is used to assess the model's quality with every parameter combination and the best model configuration is chosen. The test data has to be disjoint from the training data. We can directly incorporate the hyperparameter tuning into the training setup because we see two advantages. Firstly, our local models have a very low single training time, so repeating the training is not as time-consuming as it would be for NNs. This is depicted by Table 5.2, where the complete hyperparameter tuning for a GB model is still factor five faster than training a single NN. Secondly, we only have two parameters to keep track of, limiting the search space significantly. Therefore, we can still train all necessary models in seconds and reach

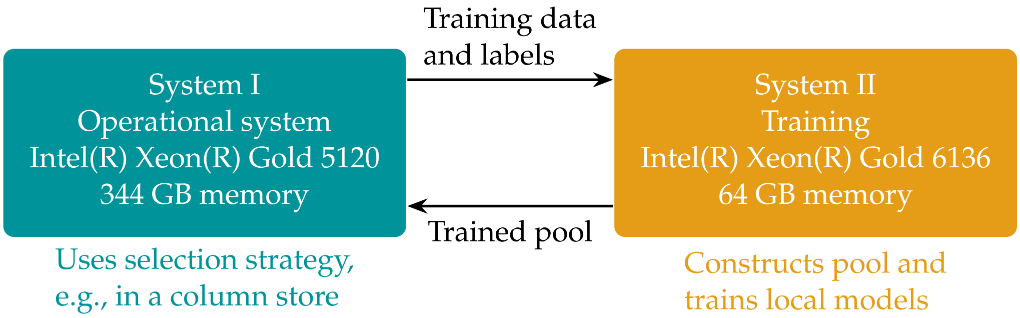


Figure 5.5: The two system setup of our integer selection strategy.

better local model qualities. Whenever we are talking about training local models for our selection strategy for integer compression from now on, we refer to a training, including the hyperparameter tuning.

To run the hyperparameter optimization for our models in the pool, we need to define the search space over the number of estimators P and the depth of decision trees d. We fix the range for the number of estimators to $[10, 50]$ with a step size of 10 and the range for the depth to $[3, 6]$. This generates a search space of $5 \cdot 4 = 20$ parameter combinations. During the hyperparameter tuning, each of the 20 configurations spawns a new training process and yields an individual model trained on the training data. Then, we select the best of those 20 models, i.e., the one with the lowest relative error on the test data. Therefore, in total, we spawn 20 training processes. However, the overall training time is still short because training is very fast for GB. From Table 5.2, a single training takes $0.12s$. So, training a model with hyperparameter tuning takes $2.4s$. This can also be shown for the forward pass. Querying a single model takes $0.8\mu s$. Therefore, querying five models for a single objective would take $4\mu s$.

In general, GB models and hyperparameter tuning are beneficial for all advantages **(A1)** to **(A4)** because of the GB architecture. As also shown for the local learning strategies for cardinality estimation and query optimizer hinting, the lightweight GB models are **(A1)** less complex and, therefore, require less memory. The GB model architecture is **(A3)** faster in training and has **(A4)** faster forward passes than, e.g., NNs. Lastly, the hyperparameter tuning and the reduced need for training data lead to **(A2)** better qualities for predicting the objectives in a pool of models.

5.7 EVALUATION

To present the qualities and advantages of our local learned selection strategy for integer compression selection, we divide the evaluation into two parts. The first part will look at the single local model's performance and how accurately it predicts the objectives. The second part details the end-to-end performance and benefits of the selection strategy.

5.7.1 Setup

Systems. All experiments are conducted on a two-system setup. System I is using the selection strategy for choosing the best algorithm for different use cases. On this operational system, we also collect the labels for the generated data to incorporate its hardware properties into the models' decisions. System I contains an Intel(R) Xeon(R) Gold 5120 with 377 GB memory. System II is an independent training system and trains the pool of models with the data generated on System I. Therefore, training does interfere with the application of the strategy or the performance of the target system. System II is comprised of an Intel(R) Xeon(R) Gold 6136 with 64 GB memory. Figure 5.5 details this two system setup and its independent application and training behavior.

Data sets. All models are always trained on a data set generated by one of our generators. For testing our models, we use two disjoint test data sets with different properties. The synthetic data from the work of Damme et al. [25] (*TODS*) contains artificially generated data with properties that are generally challenging for integer compression algorithms. The second test data set is a reduced excerpt from the *publicBI* benchmark containing real-world data from different OLAP contexts [87]. We reduced the publicBI benchmark to ca. 1GB by removing duplicated and almost equal blocks. However, we still keep data from each of the 201 example data sets within the publicBI. Additionally, we apply an order-preserving dictionary encoding to all data columns in the publicBI benchmark, which are not integers. Therefore, we are able to include the compression of floating point values and strings in a straightforward manner. With this processing step, BWHs can be applied to all data types and columns in the publicBI.

Through both data sets, we get a diverse distribution of BWHs and properties from synthetic and real-world test data where the mix of bit widths is challenging for any selection strategy. This allows us to evaluate our approaches over a diverse set of columns and data distributions.

Algorithms. Throughout the evaluation, we consider five state-of-the-art integer compression algorithms. All algorithm implementations are vectorized using AVX-512.

> *Static bit packing (static bp)* represents all integers in a data set using the number of bits required for the largest value. This algorithm is easy to implement but cannot adapt to varying data distribution.
>
> *Dynamic bit packing (dynamic bp)* divides a data set into blocks of 512 values each and represents all values in the block with the bit width of the block's largest value, i.e., an individual bit width is chosen per block.
>
> *Group simple* packs as many data elements as possible into a 512-bit vector register, whereby a common bit width is chosen for a variable number of data elements.
>
> *Delta encoding (DELTA)* only saves the differences between each successive integer value in the data.
>
> *Frame of reference (FOR)* also saves a difference instead of data values. However, in contrast to DELTA, it calculates the differences from a fixed reference point for each value.

Because we train five times five models, one for each algorithm-objective combination, our pool contains 25 models in total.

Table 5.3: Relative error for objectives (avg. over all algorithms).

Generator	compr ram2ram	decompr ram2ram	compr cache2ram	decompr ram2reg	compr rate
la-ola	11.81%	2.32%	17.80%	19.20%	12.84%
outliers	13.70%	1.23%	17.48%	16.82%	16.74%
tidal	**6.03%**	**1.17%**	**7.68%**	**6.87%**	**7.05%**

Table 5.4: Adding new algorithms to the pool only marginally affects performance.

Algorithm	Training time per model [s]	Training time total [s]	Decision time per objective [μs]	Decision time total [μs]
static bp	2.4	2.4	0.8	4.0
+dynamic bp	2.4	4.8	1.6	8.0
+group simple	2.4	7.2	2.4	12.0
+DELTA	2.4	9.6	3.2	16.0
+FOR	2.4	12.0	4.0	20.0

5.7.2 Single Model Evaluation

In this section, we present the qualities of the local models to predict the objectives. For this, we need to define the relative error between the predictions of a model $\hat{f}_a$ and the true objective values f_a for an algorithm a. For a model on a data set X consisting of $|X|= n$ data points, the relative error is defined as:

$$rel(a, X) = \frac{200\%}{n} \sum_{i=1}^{n} \frac{|\hat{f}_a(X_i) - f_a(X_i)|}{|\hat{f}_a(X_i)|+|f_a(X_i)|} \tag{5.1}$$

The relative error gives the same comparable assessment of quality over all models even though the ranges of the objectives vary. An absolute error would fail such a comparison. The lower the relative error, the better the model is performing.

Table 5.3 shows the qualities of all models per objective. Training is done on the user-independent data from our generators and the errors are retrieved on the TODS test data set. Therefore, training and test data are disjoint. The prediction errors are averaged over the models for the five different algorithms per objective. The overall quality of our local models for all objectives is good because no model produces a higher error than 20%. We use this evaluation to additionally compare our three generators. The tidal generator outperforms all other generators. This means the generation idea behind it produces the most representative training data for all objectives and algorithms. With this qualitative evaluation, we demonstrate that the advantage **(A1)** less complex models for the local selection strategy is reached by specializing the local models to an algorithm-objective combination. Therefore, small GB models are capable of expressing the problem space of integer compression selection.

We also have a look at the runtimes of our models regarding training and forward passes. Our local selection strategy is by design extendable as presented in Section 5.3. Adding new algorithms adds overhead to the training and application process, but Table 5.4 shows that this is negligible. Every new algorithm to the pool only adds linearly to the training and forward pass times. In total, we are faster with a decision over all objectives with the local models than a global model for one objective, i.e., CodecDB with $56\mu s$ (cf. Table 5.2). Therefore, the extendable divide-and-conquer approach is feasible in a way that it is faster than other approaches during training and decision-making. Therefore, we show that the advantages **(A3)** faster training and **(A4)** faster forward passes hold true for our local selection strategy. Through the smaller models in the pool, we can reduce training times and forward passes and outperform other ML-based global models.

5.7.3 Selection Strategy Evaluation

Up until now, we only compared the single model performance of our local models. However, a selection strategy for integer compression should also produce good qualities in choosing the right algorithms for a data set. To assess the quality of the local learned selection strategy, we need some indicators. We use two metrics for quality evaluation. The first metric is accuracy: Out of $|X|= n$ samples, how many times did the strategy predict the correct algorithm?

$$acc_A(X) = \frac{true\,positives}{n} \tag{5.2}$$

The higher the accuracy, the better the selection strategy because the amount of correct predictions is higher. The second indicator is the slowdown. This metric shows how much performance we lose due to misclassification. If our strategy does not select the correct algorithm, we want to quantify the loss we get for any objective. We argue that misclassifications and a low accuracy are not severe if the slowdown is low. For a single sample x, the slowdown is defined as the Symmetric Absolute Percentage Error (SAPE) between the true objective value of the best algorithm b as $f_b(x)$ and the true objective value of the chosen algorithm a as $f_a(x)$.

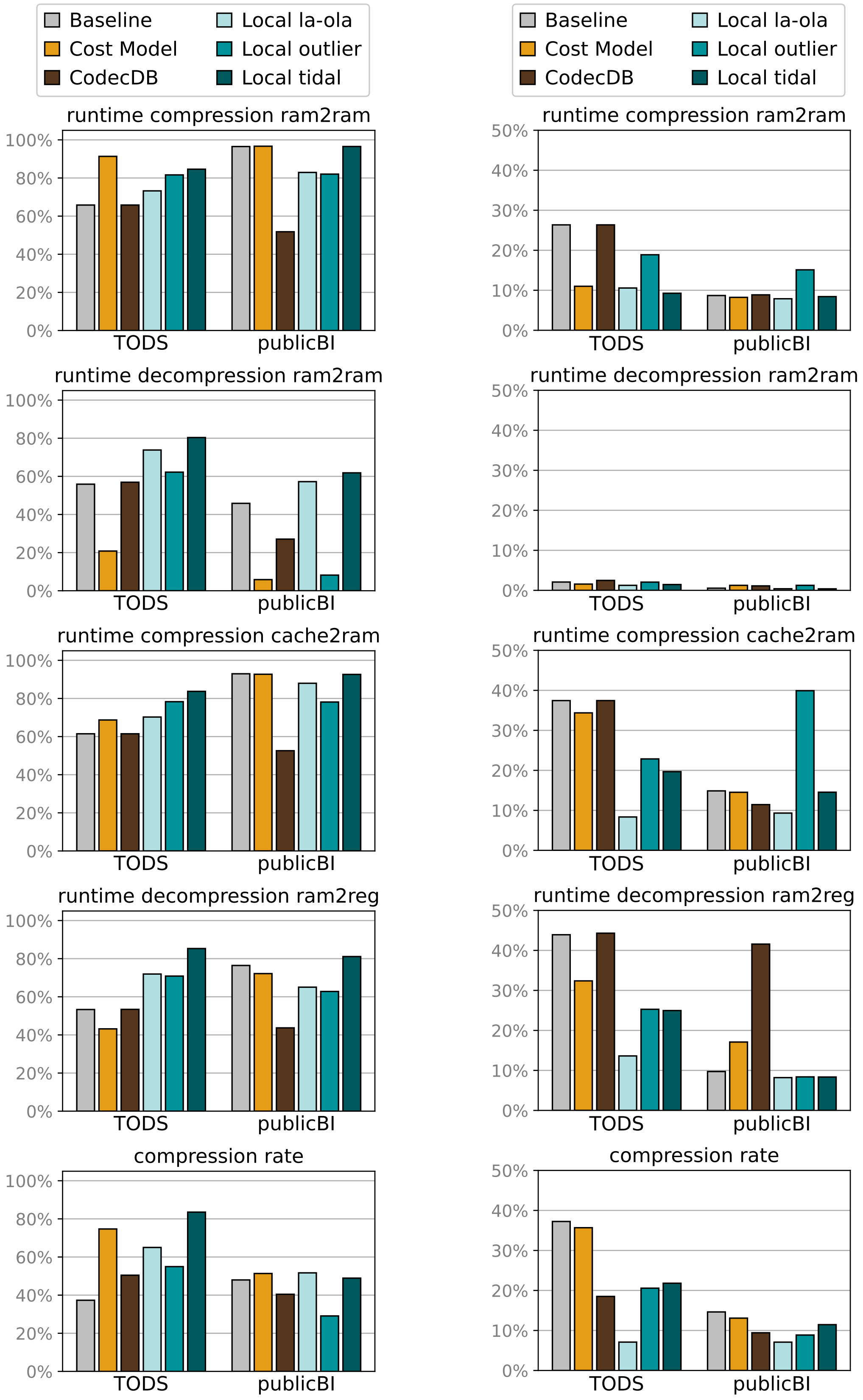
$$\mathit{SAPE}(b, a, x) = 200\% \cdot \frac{|f_a(x) - f_b(x)|}{|f_a(x)|+|f_b(x)|} \tag{5.3}$$

The slowdown is only defined for $a \neq b$. For a complete data set X, we require the best (B) and chosen algorithms (A) for all samples and average the SAPE to receive the Symmetric Mean Absolute Percentage Error (SMAPE) for the slowdown.

$$\mathit{SMAPE}(B, A, X) = \frac{200\%}{n} \cdot \sum_{i=1}^{n} \frac{|f_{A_i}(X_i) - f_{B_i}(X_i)|}{|f_{A_i}(X_i)|+|f_{B_i}(X_i)|} \tag{5.4}$$

The lower the slowdown, the better because we lose less percental performance for a certain objective. Both metrics work for all objectives. With these metrics, we can compare the overall quality of the selection strategy.

To show the quality of our approach, we compare it to two classical approaches and one learned global model approach. The first traditional approach is a *baseline* method, which chooses the simplest algorithm *static bit packing* for each block we want to compress, whereby the block size is 2,048 values in all experiments. The next traditional approach is the complex cost model introduced by Damme et al. [25]. The last one is the classification-based global model from CodecDB [40]. This model allows us to directly compare the benefits of a local model approach to a state-of-the-art ML-based model. We extend the CodecDB model to include the additional objectives O2 to O5, which were not included in the original paper. This is done by generating one global classification model for every objective. To evaluate our approach properly, we use both the validation (TODS) and test data (publicBI) and report the accuracy and slowdown for all six selection strategies.



(a) Accuracy (higher is better).

(b) Slowdown (lower is better).

Figure 5.6: Overall quality of the local selection strategy.

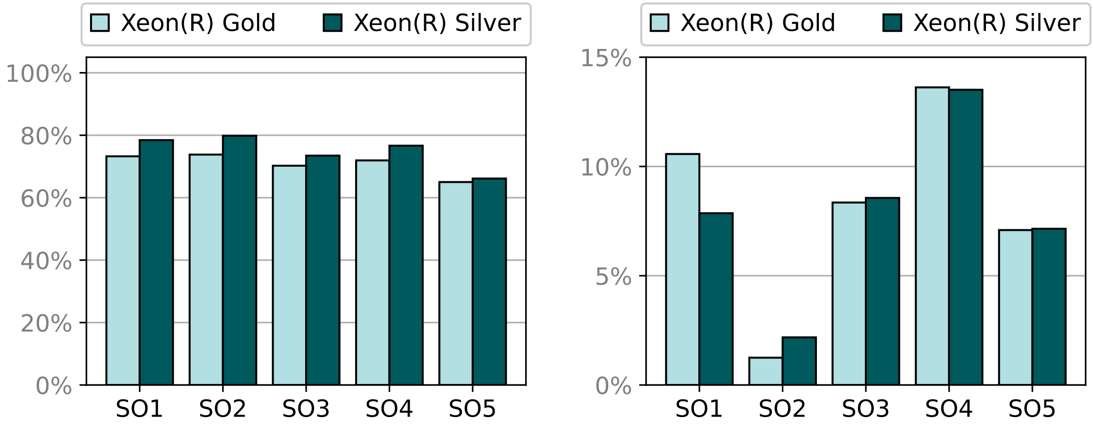
Figure 5.6 compares all related works and our local selection strategy based on one of the three generators *Local la-ola*, *Local outlier*, or *Local tidal*.

From the accuracy evaluation in Figure 5.6a, we can derive that our local learned selection strategy with the tidal generator produces the best (highest) results. It beats its competitors in most cases, except for the compression rate for the publicBI benchmark. Even then, the other approaches are only marginally better and our local selection strategy with the la-ola generator is best. The global model from CodecDB suffers from overfitting and, therefore, does not generalize to the test data. In general, the accuracy of choosing the right algorithm for all data blocks varies widely across the board of techniques. Contrarily, the slowdown evaluation in Figure 5.6b shows one common best technique. Here, the la-ola generator outperforms all other approaches by yielding the smallest slowdowns. The traditional approaches detail a very high slowdown. Even with their high accuracy, the use of these approaches would severely reduce performance. The global model shows qualities between the traditional approaches and our local models with a wide range of measured slowdowns. This is consistent with the accuracy evaluation and is again an indicator of overfitting and missing model expressiveness compared to our local models.

All in all, we derive the general recommendation that the tidal generator performs better in terms of accuracy and the la-ola generator is better for the slowdown. This means that with the tidal generator, our local learned selection strategy chooses the right algorithm more often, but if it misclassifies, the resulting slowdown is worse. For a more comprehensive overview of the data presented for all approaches in this section, we highlight all results in Table A.1 in the appendix. This extensive evaluation demonstrates that advantage **(A2)** better quality of a local selection strategy solves the corresponding shortcoming. The local models can use their expressiveness to concentrate on a certain sub-problem of predicting the best compression algorithm. Here, a sub-problem is the direct prediction of the objective for a certain algorithm. A global model has to model every algorithm and its properties at once and, therefore, loses quality.

5.7.4 Different Hardware Setups

Lastly, we detail the influence of hardware properties on our predictions. Besides data properties, the hardware properties are very important for the integer compression algorithms because they can directly influence their behavior [24]. Our approach models the hardware implicitly, meaning we do not feed features concerning the hardware to the model. The hardware behavior is modeled via the label collection. There, the compression algorithm discloses the hardware properties by producing different objective measurements on different hardware. The different measurements impact the training of the local models in a way that they can pick up different hardware properties. To test if our local learning strategy is able to adapt to different hardware properties, we update our experimental setup. We replace System I containing an Intel(R) Xeon(R) Gold 5120 and 377GB memory with an Intel(R) Xeon(R) Silver 4214R and 125GB memory. System II is kept the same for model training. Figure 5.7 shows the accuracy and slowdown for all objectives. Whereas the accuracy is always better on the new hardware, the slowdown is better for four out of five objectives. However, overall, the local selection strategy works on the new hardware as well (or even better) than on the previously used hardware. We argue that the improvement of our strategy on the new hardware is because the Intel(R) Xeon(R) Silver 4214R is from a newer generation of CPUs. Therefore, its performance is expected to be better, i.e., more stable and easier to predict.



(a) Accuracy (higher is better).

(b) Slowdown (lower is better).

Figure 5.7: The local selection strategy for compression on different hardware.

5.8 SUMMARY

In this chapter, we presented a local learned selection strategy for integer compression. Besides the inner workings of our approach, we show the four advantages as defined in Section 5.3.

(A1) Lesser model complexity by specializing to a sub-problem, i.e., a specific algorithm-objective combination

(A2) Better qualities with more expressive local models concentrating on the algorithms' properties and also being extendable by design

(A3) Faster training times by using more efficient GB models

(A4) Faster forward passes from less complex local GB models

We argue that local models improve the quality and performance of integer compression selection for different use cases. The application of our learned selection strategy is not limited to one scenario, like query processing, but can also be used in compression recommendation. We have shown the qualities and advantages extensively throughout this chapter. Comparing it to traditional and ML-based approaches, we demonstrated the potential of our approach. The combination of the extendable pool of models and the implicit hardware modeling discloses our local learning strategy's high degree of flexibility compared to the fixed architectures of global models. For example, one can use any ML model as a local model in the pool, not just GB models. Additionally, one can also use different ML models for every single algorithm-objective combination to optimize the selection strategy regarding the requirements of such a combination. Generally, a local learning strategy is beneficial for ML-based compression selection according to quality and performance especially compared to global models.

6

VISION AND BEST PRACTICES OF GENERALIZED LOCAL LEARNING STRATEGIES

The local learning strategy is not fixed to solve one problem. It is a flexible approach that can be used in different areas. We demonstrated that by applying it to three use cases or applications. These are cardinality estimation, query optimizer hinting, and integer compression selection. All these applications are very dissimilar, but a local learning strategy positively influences each and every one of them. This is not a given thing because modifying an ML approach to certain DBMS components is complex and there is no specific process for it. Even though one work claims to unify the process of learning problems in DBMS [97], it falls short to generalize any problem. Additionally, the manual effort to apply this unified model to a new problem is still high. Furthermore, its characteristic makes the unifying model opaque and its decisions are not traceable.

To help understand the problems of ML in DBMS, we collect the similarities of all three of our applications and try to derive some recommendations and prerequisites for a standardized process and platform for local learning strategies.

First and foremost, the ML problems in DBMSs are complex and hard to solve. However, most of these troubles stem from the lack of describing data or *meta data* for the database system. Whereas DBMS are functional data storages providing benefits to a lot of data-driven use cases, their meta data handling lacks behind in most cases. This leads to several problems with ML models because these models build a lot of their expressiveness upon the meta-information of the data. For example, a learned cardinality estimator needs the ranges of all columns to locate predicates within these ranges. Here, the meta-information about column ranges is important, among other things, to implicitly model minimum and maximum values of possible predicate values in queries. Generally speaking, the storing and usability of meta data in DBMS is not on par for the comfortable deployment of ML models. This is true not only for local models but for global models as well. As Andy Pavlo, CEO of OtterTune, said during a talk about the greatest challenges of ML for DBMS: "The problem is not that ML does not work in a DBMS but that it is hard to wrangle the information from the database system." Some DBMS, like Snowflake, already provide a meta storage [82], but their motivation is reporting DBMS performance statistics rather than data collection and preparation for ML. This is a good starting argument for analyzing how DBMS should change to support better ML components. In the following, we will list the further requirements identified during our work.

6.1 PROBLEM SPACE DEFINITION

As mentioned in Chapter 2, local and global models base their expressiveness on an underlying problem space. Usually, these problem spaces are enormous and need to be reduced to a specific use case. For cardinality estimation and query hinting, the problem space is the combination of every possible underlying schema. The schemas are defined by entity-relationship diagrams. Therefore, a standardized definition exists. However, due to the flexibility in SQL query syntax, the advantages of such a standardized schema design are negated. This can be circumvented by transforming queries to QEPs with a parser. However, this does not suffice for other problems, like the raw data-driven integer compression selection problem. For integer algorithm compression, the problem space is the collection of all variable-length integer data arrays. To represent this problem space, we need different data combinations to be shown to the model. We reduced this particular problem space size by using bit width (bw) histograms. So, in general, **we recommend using data representations or abstractions to reduce problem space complexity**, like schemas and QEPs for workload-driven problem space definitions or bw histograms for raw data-based problems. However, no general process exists to find the best data representation for a problem. Therefore, a lot of human knowledge and expertise is needed to model a problem space in a DBMS. Even defining a problem space is not trivial and relies on the user′s needs, which can vary broadly.

6.2 TRAINING DATA GENERATION

Another big issue that occurred in all three applications is the lack of training data. Without training data, neither local nor global models can produce meaningful results. There are two possible ways to collect training data: by collecting user data or by generating artificial representative data. The first approach is usually difficult to implement because of privacy or intellectual property concerns. Therefore, most research resorts to data generation [36, 44, 47]. The challenge is to generate as much meaningful data as possible but at as little as is needed to train high-quality models. This balance is hard to produce because the evaluation measurement for the data generation is the final performance of the local models and how well they solve the problem at hand. For cardinality estimation and optimizer hinting, we could use representative user data from the Stack, JOB, and TPC-H benchmarks. However, because they are workload-driven benchmarks, they fail to be beneficial for the integer compression selection. There, we had to use a generated data set because the user data set publicBI did not produce models generalizing to other data. From the extent of our analysis of different generators, it is clear that the data generation is also a problem-specific task to be addressed for each problem individually. Therefore, **we recommend using pipelines where the data generation technique can be changed easily and its effect on the final quality of the ML component can be reported directly.** This can lead to accelerated engineering of data generators for local models as ML-based DBMS components.

6.3 FEATURE ENGINEERING

Feature engineering is one of the most complex parts of ML modeling [76]. As shown in Chapter 5 for integer compression selection, the overhead of (human) feature engineering is only solved in two possible ways: fixed feature selection or the generation and testing of all combinations of features and their influences. The latter also includes the different ways of derived features, like averages or minimum and maximum. Both approaches have their advantages and disadvantages. The fixed feature sets are easy to calculate and add little overhead to setting up a local model strategy. However, they might not represent all data properties, leading to sub-optimal models in return. The generation of all possible features does not have this problem because it is able to test feature combinations until the best local models have been found. Nonetheless, the generation can run arbitrarily long without significantly improving model qualities. In a preliminary experiment for feature generation for compression selection, we improved the slowdown by only 2% after around two hours of optimization per algorithm-objective combination (30h for the whole pool of models). Therefore, for feature engineering, **we recommend using a fixed feature set derived by domain experts.** If possible, a limited pre-selection of automatically generated features can be utilized to facilitate the work of the domain experts. In our experience, domain-specific features will lead to the best local models because domain experts have an inherent understanding of their domain and only need guidance to transfer it into feature definitions.

Table 6.1: Comparison of local learning strategies for different applications.

Property	Cardinality estimation	Query optimizer hinting	Integer compression selection
Problem	regression	regression	classification
Problem space	schema	schema	integer data
Data generation	manual user dependent	manual user dependent	automatic user independent
Feature engineering	dynamic generated domain expertise	dynamic generated domain expertise	fixed derived domain expertise
Models	Neural network Gradient boosting	Gradient boosting	Gradient boosting
Quality metric	q-error	-	accuracy rel. model error
Performance metric	training times forward passes	speedups	slowdown

6.4 LOCAL MODEL TYPES

There are many ML models that can be used as local models. The focus is to use as simple and fast architectures as possible. For our three applications, we used gradient boosting (GB) in all cases. This arises from the lightweight nature of these models. The training times and forward passes of GB models are fast, while the models are very robust and expressive. In cardinality, we also used feed-forward neural networks (NN) to show that the local learning strategy is *model independent*. However, their results were not on par with GB, both in quality and performance. We tested different other models as local models during our research, like linear regression or support vector machines, but GB and NN stood out as the best. Therefore, from our experience, **we recommend the use of GB or simple NN** as local models. Nonetheless, this is not a fixed recommendation. If better models are available for a use case or in general, one should utilize these.

6.5 QUALITY AND PERFORMANCE EVALUATION

Lastly, the evaluation of ML-based approaches is not straightforward either. Firstly, there is a differentiation between quality and performance evaluation. This work strives to include both aspects and compare them to each other. However, depending on the goal of a local learning strategy, the evaluation metric can vary. This is entirely within the user's decision and can make approaches for both one use case and different use cases incomparable if different metrics are used. In cardinality estimation, we detailed the quality with the q-error and performance with runtimes of training and forward passes. For query optimizer hinting, the performance is evaluated with the workload speedups. Lastly, for integer compression selection, the quality metrics were the relative model error and the accuracy. The performance was measured via the slowdown. Due to the variety of applications and their different requirements, **we recommend an evaluation tailored to the users' needs.** This includes the most important metrics for the goals the local learning strategy has to fulfill, e.g., query runtime or memory consumption. One might also consider statistical tests for comparison, but these are also based on error or quality metrics. So, this would not solve the ambiguity of evaluating and finding the best variant.

6.6 SUMMARY

To summarize the previous findings and recommendations, we present Table 6.1, which collects the similarities and dissimilarities of our three local learning strategy applications. It is clearly visible that the applications have very different requirements for all properties. Except for including domain knowledge for feature engineering, the variation is too high to define general rules for all local learning strategies. Therefore, we argue that the generalization of local learning strategies and ML components for DBMS is complex and formulating a general process is out of the scope of this work. The need to adapt ML to the complexity of any new problem is still the most prominent problem in this area. To combat this, we propose a list of steps defining a structural analysis for local learning strategies.

1. Include domain knowledge and experts early on.
2. Define the problem space and the sub-problems (contexts). Decide if the task is a regression, classification, or reinforcement problem.
3. Use training data as representative and general as possible. If no user data is available, generate artificial data sets.
4. Engineer the features according to the gathered domain knowledge. Try to generate derived features automatically, where possible and under consideration of time limits.
5. Choose the ML models to be used as local models. Consider model complexity, training time, forward passes, and sample efficiency.
6. Find the most relevant metrics or key performance indicators that suit the application best. Distinguish between quality and performance-related aspects.
7. Consider the underlying system, e.g., a DBMS. What kind of integration should the local learning strategy undergo in this use case? How complex is the transfer to other (user) data or systems? Utilize a transparent implementation and documentation that are easy to use for others.

With this seven-step process, applying a local learning strategy to a data management problem or even problems beyond this scope can be facilitated in a structured manner. This should help others to use local learning strategies in a generalized way for their problems.

7

CONCLUSION

This dissertation presented a new way to deploy and use ML models as DBMS components in a divide-and-conquer manner. We proposed the *local learning strategy*, where large complex problems and their problem spaces are split into sub-problems and several ML models are trained to solve these specific sub-problems. This one-to-one mapping of sub-problem to ML model improves both the quality and performance of problem-solving in DBMSs. This is due to the general property of divide-and-conquer to reduce complexities. Even though divide-and-conquer is an established technique, first mentioned around 200 BC [46], we demonstrated that its application still can improve modern paradigms in Computer Science like ML. To compare our local model approach, we detailed related research, all focusing on global strategies. This focus introduces four shortcomings.

(S1) Large complex models

(S2) Overfitting and low quality

(S3) Large amounts of training data and slow training

(S4) Slow forward passes

These shortcomings negatively influence the quality and performance of global learning approaches for data management problems. However, we derived four advantages that the local learning strategy fulfills to circumvent these shortcomings and outperform global models. To evaluate our local learning strategy in detail, we focused on three important components in data management: cardinality estimation, query optimizer hinting, and integer compression algorithm selection. For all three areas, we addressed the four advantages **(A1)** to **(A4)** and how local models reach them.

(A1) Less complex models: In all three applications, we concluded that GB models are the best choice for local models. These models are lightweight and have little complexity in model architecture. This makes them faster and smaller in all use cases than other (global) models, as shown in the extensive evaluations of all three data management applications. However, our approach is independent of model architectures for local models. Our local learning approach is model-independent and any standard ML model can be used as local models. So, we were also able to solve cardinality estimation problems with simple NNs. One could even use a global model, like the MSCN, as a local model, but we would not recommend it because of the unnecessary overhead. Additionally, unlike global models in all three applications, we do not need to spend any time to find the best-fitting type of ML model architecture. For example, the MSCN for cardinality estimation or the TCN for query optimizer hinting are very specific structures tailored to one application. This limits their transferability to other problems. For optimizer hinting, queries are accelerated with higher speedups. All these improvements can be ascribed to the local models and their intelligent placement. We showed the local strategy's better qualities in comparison to baselines, traditional approaches, and global models. For example, we included PostgreSQL, MSCN [44], and DeepDB [36] for cardinality estimation or BAO [54] for query optimizer hinting. The results argue in favor of using a local learning strategy in all three use cases. Lastly, for integer compression selection, the global models modeled the decision for the best algorithm as a classification problem. This avoids transferability even within the same problem space of algorithm selection. New or modified algorithms require a completely new global model construction. The local selection strategy for integer compression and its pool of models is extendable by design, allowing direct adaption to changing environments.

- **(A2) Better quality:** The focus of the local models on sub-problems allows them to spend their full expressiveness on these sub-problems. This leads to better qualities for all three applications. For cardinality estimation, the factor of misestimates is lower. For compression selection, the best algorithm is chosen more often. For query optimizer hinting, query runtimes are faster. Overall, the qualities reached with local models are better than the ones for global models or traditional approaches.
- **(A3) Faster training:** The smaller, less complex models cannot only focus better on the sub-problems but also need less training data to learn. They can derive the specialties of a sub-problem or context with only a minimal amount of training information. This reduces the training times greatly. Most ML models' training times depend linearly on the number of training tuples. For example, cardinality estimation and query hinting models are based on training queries from workloads. So, we show that a reduced number of training queries leads to the same quality as global models or using the same amount of training queries leads to better quality for the local models. The same was verified for the integer compression selection but with different data generators for integer data instead of queries. The resulting faster deployment of local models is a key aspect. It allows to dynamically change models and contexts during runtime without interfering with the underlying system in contrast to, e.g., a reinforcement learning approach. We detailed and evaluated this with the runtime model maintenance strategy for the local learning strategy in query optimizer hinting.
- **(A4) Fast forward passes:** With the less complex models also come less complex calculations to produce a prediction per local model or a collection of local models. This helps to minimize the overhead of model applications during runtime. For all three applications, we demonstrated that the local learning strategies add less overhead to query or data processing than global models. The local models' overhead is not zero but negligible small to have an influence on system behavior or user experience. For example, for cardinality estimation, the overall overhead improvement had a factor of 1,000. For query optimizer hinting, the forward passes through a model are also in the lower microseconds, not affecting gained workload speedups received through hinting. For integer compression selection, the pool of local models for five algorithms and five objectives can be queried for the best algorithm in $20\mu s$. Overall, the small local models' combined forward passes are fast enough by orders of magnitude to not diminish the improvements they introduce to a problem.

Generally speaking, local models can balance the scale of quality and performance efficiently, meaning they can use their whole potential to improve the given use case while also adding little overhead, improving the quality or runtimes of DBMS processes. In this work, we discussed the advantages of local learning strategies for use cases directly from the data management domain. Local learning strategies can be seen as a beneficial addition to the toolbox of ML components for DBMS.

BIBLIOGRAPHY

[1] Daniel Abadi, Anastasia Ailamaki, David Andersen, Peter Bailis, Magdalena Balazinska, Philip Bernstein, Peter Boncz, Surajit Chaudhuri, Alvin Cheung, AnHai Doan, et al. The seattle report on database research. *ACM SIGMOD Record*, 48(4):44–53, 2020.

[2] Daniel Abadi, Peter A. Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. The Design and Implementation of Modern Column-Oriented Database Systems. *Found. Trends Databases*, 5(3):197–280, 2013.

[3] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 671–682. ACM, 2006.

[4] Sameet Agarwal, Rakesh Agrawal, Prasad Deshpande, Ashish Gupta, Jeffrey F. Naughton, Raghu Ramakrishnan, and Sunita Sarawagi. On the Computation of Multidimensional Aggregates. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 506–521. Morgan Kaufmann, 1996.

[5] Rakesh Agrawal, Anastasia Ailamaki, Philip A Bernstein, Eric A Brewer, Michael J Carey, Surajit Chaudhuri, AnHai Doan, Daniela Florescu, Michael J Franklin, Hector Garcia-Molina, et al. The Claremont report on database research. *ACM Sigmod Record*, 37(3):9–19, 2008.

[6] Susan Athey. The impact of machine learning on economics. In *The economics of artificial intelligence: An agenda*, pages 507–547. University of Chicago Press, 2018.

[7] Don S. Batory. B+ Trees and Indexed Sequential Files: A Performance Comparison. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data, Ann Arbor, Michigan, USA, April 29 - May 1, 1981*, pages 30–39. ACM Press, 1981.

[8] Benjamin Hilprecht and Carsten Binnig and Tiemo Bang and Muhammad El-Hindi and Benjamin Hättasch and Aditya Khanna and Robin Rehrmann and Uwe Röhm and Andreas Schmidt and Lasse Thostrup and Tobias Ziegler. DBMS Fitting: Why should we learn what we already know? In *Conference on Innovative Data Systems Research (CIDR)*, 2020.

[9] Kiran Bhageshpur. The world's most valuable resource is no longer oil, but data. https://www.economist.com/leaders/2017/05/06/the-worlds-most-valuable-resource-is-no-longer-oil-but-data, 2017. Accessed: 2023-04-19.

[10] Christopher M. Bishop. *Pattern recognition and machine learning, 5th Edition*. Information science and statistics. Springer, 2007.

[11] Martin Boissier and Max Jendruk. Workload-Driven and Robust Selection of Compression Schemes for Column Stores. In *EDBT*, pages 674–677. OpenProceedings.org, 2019.

[12] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking the memory wall in MonetDB. *Commun. ACM*, 51(12):77–85, 2008.

[13] Nicolas Bruno and Surajit Chaudhuri. Exploiting statistics on query expressions for optimization. In *SIGMOD*, pages 263–274, 2002.

[14] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. STHoles: A multidimensional workload-aware histogram. In *Proceedings of the 2001 ACM International Conference on Management of Data (SIGMOD)*, pages 211–222, 2001.

[15] Lujing Cen, Andreas Kipf, Ryan Marcus, and Tim Kraska. LEA: A Learned Encoding Advisor for Column Stores. In *Fourth Workshop in Exploiting AI Techniques for Data Management*, aiDM '21, page 32–35, New York, NY, USA, 2021. Association for Computing Machinery.

[16] Surajit Chaudhuri. An Overview of Query Optimization in Relational Systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA*, pages 34–43. ACM Press, 1998.

[17] Surajit Chaudhuri. Query optimizers: time to rethink the contract? In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 961–968. ACM, 2009.

[18] Surajit Chaudhuri, Umeshwar Dayal, and Vivek R. Narasayya. An overview of business intelligence technology. *Commun. ACM*, 54(8):88–98, 2011.

[19] Jie Chen, Kees de Hoogh, John Gulliver, Barbara Hoffmann, Ole Hertel, Matthias Ketzel, Mariska Bauwelinck, Aaron van Donkelaar, Ulla A. Hvidtfeldt, Klea Katsouyanni, Nicole A.H. Janssen, Randall V. Martin, Evangelia Samoli, Per E. Schwartz, Massimo Stafoggia, Tom Bellander, Maciek Strak, Kathrin Wolf, Danielle Vienneau, Roel Vermeulen, Bert Brunekreef, and Gerard Hoek. A comparison of linear regression, regularization, and machine learning algorithms to develop Europe-wide spatial models of fine particles and nitrogen dioxide. *Environment International*, 130:104934, 2019.

[20] Yu Chen and Ke Yi. Two-level sampling for join size estimation. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*, pages 759–774, 2017.

[21] Zhiyuan Chen, Johannes Gehrke, and Flip Korn. Query Optimization In Compressed Database Systems. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, pages 271–282. ACM, 2001.

[22] Transaction Processing Council. Tpc-h. https://www.tpc.org/tpch/, 2023. Accessed: 2023-05-25.

[23] Patrick Damme, Marius Birkenbach, Constantinos Bitsakos, Matthias Boehm, Philippe Bonnet, Florina Ciorba, Mark Dokter, Pawl Dowgiallo, Ahmed Eleliemy, Christian Faerber, et al. DAPHNE: An Open and Extensible System Infrastructure for Integrated Data Analysis Pipelines. In *Conference on Innovative Data Systems Research (CIDR)*, 2022.

[24] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses). In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*, pages 72–83. OpenProceedings.org, 2017.

[25] Patrick Damme, Annett Ungethüm, Juliana Hildebrandt, Dirk Habich, and Wolfgang Lehner. From a Comprehensive Experimental Survey to a Cost-based Selection Strategy for Lightweight Integer Compression Algorithms. *ACM Trans. Database Syst.*, 44(3):9:1–9:46, 2019.

[26] Patrick Damme, Annett Ungethüm, Johannes Pietrzyk, Alexander Krause, Dirk Habich, and Wolfgang Lehner. MorphStore: Analytical Query Engine with a Holistic Compression-Enabled Processing Model. *Proc. VLDB Endow.*, 13(11):2396–2410, 2020.

[27] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. Selectivity estimation for range predicates using lightweight models. *Proceedings of the VLDB Endowment*, 12(9):1044–1057, 2019.

[28] Jessica Maria Echterhoff, Juan Haladjian, and Bernd Brügge. Gait and jump classification in modern equestrian sports. In *Proceedings of the 2018 ACM International Symposium on Wearable Computers, UbiComp 2018, Singapore, Singapore, October 8-12, 2018*, pages 88–91. ACM, 2018.

[29] Franz Faerber, Alfons Kemper, Per-Åke Larson, Justin J. Levandoski, Thomas Neumann, and Andrew Pavlo. Main Memory Database Systems. *Found. Trends Databases*, 8(1-2):1–130, 2017.

[30] Pit Fender and Guido Moerkotte. A new, highly efficient, and easy to implement top-down join enumeration algorithm. In *ICDE*, pages 864–875, 2011.

[31] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Discrete Mathematics and Theoretical Computer Science*, pages 137–156. Discrete Mathematics and Theoretical Computer Science, 2007.

[32] Murray Gell-Mann. Complex adaptive systems. *Metaphors, Models, and Reality, Proc*, pages 17–45, 1994.

[33] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. Compressing Relations and Indexes. In *Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Florida, USA, February 23-27, 1998*, pages 370–379. IEEE Computer Society, 1998.

[34] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997.

[35] Axel Hertzschuch, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. Turbo-Charging SPJ Query Plans with Learned Physical Join Operator Selections. *Proc. VLDB Endow.*, 15(11):2706–2718, sep 2022.

[36] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. DeepDB: Learn from Data, not from Queries! *Proceedings of the VLDB Endowment*, 13(7):992–1005, 2020.

[37] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press, 1992.

[38] Jing Huang and Matthew Perry. A semi-empirical approach using gradient boosting and k-nearest neighbors regression for gefcom2014 probabilistic solar power forecasting. *International Journal of Forecasting*, 32(3):1081–1086, 2016.

[39] IBM. IBM Db2: Histograms in workload management. https://www.ibm.com/support/producthub/db2/docs/content/SSEPGG_11.5.0/com.ibm.db2.luw.admin.wlm.doc/doc/c0052789.html, 2023. Accessed: 2023-04-19.

[40] Hao Jiang, Chunwei Liu, John Paparrizos, Andrew A. Chien, Jihong Ma, and Aaron J. Elmore. Good to the Last Bit: Data-Driven Encoding with CodecDB. In *SIGMOD Conference*, pages 843–856. ACM, 2021.

[41] Yingting Jin, Yuzhuo Fu, Ting Liu, and Lan Dong. Adaptive Compression Algorithm Selection Using LSTM Network in Column-oriented Database. In *2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, pages 652–656. IEEE, 2019.

[42] Alfons Kemper and André Eickler. *Datenbanksysteme - Eine Einführung, 8. Auflage*. Oldenbourg, 2011.

[43] Andreas Kipf. Learned Cardinalities in PyTorch. https://github.com/andreaskipf/learnedcardinalities, 2019. Accessed: 2023-03-20.

[44] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *Conference on Innovative Data Systems Research (CIDR)*. www.cidrdb.org, 2019.

[45] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. RadixSpline: a single-pass learned index. In *aiDM@SIGMOD*, pages 5:1–5:5. ACM, 2020.

[46] Donald Ervin Knuth. *The Art of Computer Programming: Volume 3, Sorting and Searching, 2nd Edition*. Addison-Wesley, 1998.

[47] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 489–504. ACM, 2018.

[48] Wolfgang Lehner. *Datenbanktechnologie für Data-Warehouse-Systeme. Konzepte und Methoden*. dpunkt.verlag, 2003.

[49] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *PVLDB*, 9(3):204–215, 2015.

[50] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *VLDB J.*, 27(5):643–668, 2018.

[51] Daniel Lemire and Leonid Boytsov. Decoding billions of integers per second through vectorization. *Softw., Pract. Exper.*, 45(1):1–29, 2015.

[52] M. Lichman. Forest CoverType dataset. https://archive.ics.uci.edu/ml/datasets/covertype, 2013. Accessed: 2023-03-20.

[53] Henry Liu, Mingbin Xu, Ziting Yu, Vincent Corvinelli, and Calisto Zuzarte. Cardinality estimation using neural networks. In *CASCON*, pages 53–59. IBM / ACM, 2015.

[54] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Making Learned Query Optimization Practical. *SIGMOD Rec.*, 51(1):6–13, jun 2022.

[55] Ryan Marcus and Olga Papaemmanouil. Deep Reinforcement Learning for Join Order Enumeration. In *aiDM@SIGMOD*, pages 3:1–3:4. ACM, 2018.

[56] Ryan C. Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.*, 12(11):1705–1718, 2019.

[57] Volker Markl, Vijayshankar Raman, David E. Simmen, Guy M. Lohman, and Hamid Pirahesh. Robust Query Processing through Progressive Optimization. In *SIGMOD*, pages 659–670, 2004.

[58] Llew Mason, Jonathan Baxter, Peter L. Bartlett, and Marcus R. Frean. Boosting Algorithms as Gradient Descent. In *Advances in Neural Information Processing Systems 12, NIPS Conference, Denver, Colorado, USA, November 29 - December 4, 1999*, pages 512–518. The MIT Press, 1999.

[59] Holger J. Meyer, Hannes Grunert, Tim Waizenegger, Lucas Woltmann, Claudio Hartmann, Wolfgang Lehner, Mahdi Esmailoghli, Sergey Redyuk, Ricardo Martinez, Ziawasch Abedjan, Ariane Ziehn, Tilmann Rabl, Volker Markl, Christian Schmitz, Dhiren Devinder Serai, and Tatiane Escobar Gava. Particulate Matter Matters - The Data Science Challenge @ BTW-2019. *Datenbank-Spektrum*, 19(3):165–182, 2019.

[60] Guido Moerkotte, David DeHaan, Norman May, Anisoara Nica, and Alexander Böhm. Exploiting ordered dictionaries to efficiently construct histograms with q-error guarantees in SAP HANA. In *Proceedings of the 2014 ACM International Conference on Management of Data (SIGMOD)*, pages 361–372, 2014.

[61] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. Preventing bad plans by bounding the impact of cardinality estimation errors. *Proceedings of the VLDB Endowment*, 2(1):982–993, 2009.

[62] Ravi Mukkamala and Sushil Jajodia. A Note on Estimating the Cardinality of the Projection of a Database Relation. *ACM Trans. Database Syst.*, 16(3):564–566, 1991.

[63] Magnus Müller, Lucas Woltmann, and Wolfgang Lehner. Enhanced Featurization of Queries with Mixed Combinations of Predicates for ML-based Cardinality Estimation. *Proceedings of the 26th International Conference on Extending Database Technology*, page 273–284, 2023.

[64] MySQL. Optimizer Hints. https://dev.mysql.com/doc/refman/8.0/en/optimizer-hints.html, 2023. Accessed: 2023-05-05.

[65] United States. Executive Office of the President. Artificial intelligence, automation, and the economy. Technical report, The White House, 2016.

[66] OpenAI. ChatGPT: Optimizing Language Models for Dialogue. https://openai.com/blog/chatgpt/, 2022. Accessed: 2023-04-19.

[67] Oracle. Database SQL Tuning Guide: Histograms. https://docs.oracle.com/database/121/TGSQL/tgsql_histo.htm#TGSQL366, 2023. Accessed: 2023-04-19.

[68] Oracle. Optimizer Hints. https://docs.oracle.com/cd/B12037_01/server.101/b10752/hintsref.htm, 2023. Accessed: 2023-05-05.

[69] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. An Empirical Analysis of Deep Learning for Cardinality Estimation. *arXiv:1905.06425*, 2019.

[70] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. Self-Driving Database Management Systems. In *Conference on Innovative Data Systems Research (CIDR)*, 2017.

[71] Robert Peharz, Antonio Vergari, Karl Stelzner, Alejandro Molina, Martin Trapp, Xiaoting Shao, Kristian Kersting, and Zoubin Ghahramani. Random Sum-Product Networks: A Simple and Effective Approach to Probabilistic Deep Learning. In *Proceedings of the Thirty-Fifth Conference on Uncertainty in Artificial Intelligence, UAI 2019, Tel Aviv, Israel, July 22-25, 2019*, volume 115 of *Proceedings of Machine Learning Research*, pages 334–344. AUAI Press, 2019.

[72] Hoifung Poon and Pedro M. Domingos. Sum-Product Networks: A New Deep Architecture. In *UAI 2011, Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence, Barcelona, Spain, July 14-17, 2011*, pages 337–346. AUAI Press, 2011.

[73] PostgreSQL. Query Planning. https://www.postgresql.org/docs/current/runtime-config-query.html, 2023. Accessed: 2023-05-05.

[74] Viktor Rosenfeld, Max Heimel, Christoph Viebig, and Volker Markl. The Operator Variant Selection Problem on Heterogeneous Hardware. In *ADMS@VLDB*, pages 1–12, 2015.

[75] Mark A. Roth and Scott J. Van Horn. Database Compression. *SIGMOD Rec.*, 22(3):31–39, 1993.

[76] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson, 2020.

[77] Florin Rusu and Alin Dobra. Sketches for size of join estimation. *ACM Transactions on Database Systems (TODS)*, 33(3):1–46, 2008.

[78] Maximilian E. Schüle, Luca Scalerandi, Alfons Kemper, and Thomas Neumann. Blue Elephants Inspecting Pandas: Inspection and Execution of Machine Learning Pipelines in SQL. In *Proceedings 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28-31, 2023*, pages 40–52. OpenProceedings.org, 2023.

[79] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *SIGMOD*, SIGMOD '79, page 23–34, New York, NY, USA, 1979. Association for Computing Machinery.

[80] SQL Server. Hints (Transact-SQL) - Query. https://learn.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql-query?view=sql-server-ver16, 2023. Accessed: 2023-05-05.

[81] Michael Shekelyan, Anton Dignös, and Johann Gamper. Digithist: a histogram-based data summary with tight error bounds. *Proceedings of the VLDB Endowment*, 10(11):1514–1525, 2017.

[82] Snowflake. Understanding Your Snowflake Utilization, Part 1: Warehouse Profiling, 2017. Accessed: 2023-06-01.

[83] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-Store: A Column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 553–564. ACM, 2005.

[84] Nils Strassenburg, Dominic Kupfer, Julia Kowal, and Tilmann Rabl. Efficient Multi-Model Management. In *Proceedings 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28-31, 2023*, pages 457–463. OpenProceedings.org, 2023.

[85] Immanuel Trummer, Junxiong Wang, Ziyun Wei, Deepak Maram, Samuel Moseley, Saehan Jo, Joseph Antonakakis, and Ankush Rayabhari. SkinnerDB: Regret-bounded Query Evaluation via Reinforcement Learning. *ACM Trans. Database Syst.*, 46(3):9:1–9:45, 2021.

[86] Robert Ulbricht, Ulrike Fischer, Lars Kegel, Dirk Habich, Hilko Donker, and Wolfgang Lehner. ECAST: A Benchmark Framework for Renewable Energy Forecasting Systems. In *Proceedings of the Workshops of the EDBT/ICDT 2014 Joint Conference (EDBT/ICDT 2014), Athens, Greece, March 28, 2014*, volume 1133 of *CEUR Workshop Proceedings*, pages 148–155. CEUR-WS.org, 2014.

[87] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Mühlbauer, Thomas Neumann, and Manuel Then. Get Real: How Benchmarks Fail to Represent the Real World. In *Proceedings of the 7th International Workshop on Testing Database Systems, DBTest@SIGMOD 2018, Houston, TX, USA, June 15, 2018*, pages 1:1–1:6. ACM, 2018.

[88] TaiNing Wang and Chee-Yong Chan. Improved correlated sampling for join size estimation. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 325–336. IEEE, 2020.

[89] Lucas Woltmann, Patrick Damme, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. Learned Selection Strategy for Lightweight Integer Compression Algorithms. In *Proceedings 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28-31, 2023*, pages 552–564. OpenProceedings.org, 2023.

[90] Lucas Woltmann, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. Best of both worlds: combining traditional and machine learning models for cardinality estimation. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2020*, pages 4:1–4:8. ACM, 2020.

[91] Lucas Woltmann, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. Aggregate-based Training Phase for ML-based Cardinality Estimation. *Datenbank-Spektrum*, 22(1):45–57, 2022.

[92] Lucas Woltmann, Claudio Hartmann, Wolfgang Lehner, Paul Rausch, and Katja Ferger. Sensor-based jump detection and classification with machine learning in trampoline gymnastics. *German Journal of Exercise and Sport Research*, pages 1–9, 2022.

[93] Lucas Woltmann, Claudio Hartmann, Maik Thiele, Dirk Habich, and Wolfgang Lehner. Cardinality estimation with local deep learning models. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2019*, pages 1–8. ACM, 2019.

[94] Lucas Woltmann, Dominik Olwig, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. PostCENN: PostgreSQL with Machine Learning Models for Cardinality Estimation. *Proc. VLDB Endow.*, 14(12):2715–2718, 2021.

[95] Lucas Woltmann, Jerome Thiessat, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. FASTgres: Making Learned Query Optimizer Hinting Effective. *Proc. VLDB Endow.*, 2023. Accepted for publication. Unpublished.

[96] Lucas Woltmann, Peter Benjamin Volk, Michael Dinzinger, Lukas Gräf, Sebastian Strasser, Johannes Schildgen, Claudio Hartmann, and Wolfgang Lehner. Data Science Meets High-Tech Manufacturing - The BTW 2021 Data Science Challenge. *Datenbank-Spektrum*, 22(1):5–10, 2022.

[97] Ziniu Wu, Pei Yu, Peilun Yang, Rong Zhu, Yuxing Han, Yaliang Li, Defu Lian, Kai Zeng, and Jingren Zhou. A Unified Transferable Model for ML-Enhanced DBMS. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org, 2022.

[98] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. Balsa: Learning a Query Optimizer Without Expert Demonstrations. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 931–944. ACM, 2022.

[99] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, and Ion Stoica. Deep Unsupervised Cardinality Estimation. *Proc. VLDB Endow.*, 13(3):279–292, 2019.

[100] Xiang Yu, Chengliang Chai, Guoliang Li, and Jiabin Liu. Cost-based or Learning-based? A Hybrid Query Optimizer for Query Plan Selection. *Proc. VLDB Endow.*, 15(13):3924–3936, 2022.

[101] Bin Yuan, M. M. Kamruzzaman, and Shaonan Shan. Application of Motion Sensor Based on Neural Network in Basketball Technology and Physical Fitness Evaluation System. *Wirel. Commun. Mob. Comput.*, 2021:5562954:1–5562954:11, 2021.

[102] Ji Zhang. AlphaJoin: Join Order Selection à la AlphaGo. In *PhD@VLDB*, volume 2652 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2020.

LIST OF FIGURES

1.1 The interplay of DBMSs and ML. 14

2.1 Coverage of problem spaces by different functions f_c and f_m. 20

3.1 Schema for the example query. 26
3.2 Process for cardinality estimation with learned models. 27
3.3 The multi-set convolutional neural network (MSCN) architecture. 30
3.4 The relational sum-product network (RSPN) for columns `R.A` and `T.D`. . . . 32
3.5 Global and local approach contexts and their model complexities. 33
3.6 Coverage of problem spaces by global f_g and local models f_l. 34
3.7 Global and local approach contexts and their placement. 35
3.8 Local model structures used for cardinality estimation. 38
a A feed-forward neural network architecture. 38
b A gradient boosting model architecture. 38
3.9 The aggregate-based labeling with grouping sets. 44
3.10 The trade-off between quality and training times according to number of training queries. 48

4.1 The structure of a traditional query optimizer with hints. 54
4.2 The QEP changes if nested loop joins are prohibited via hints. 55
4.3 The Bandit Optimizer (BAO) for learned query optimizer hinting. 57
4.4 The local learning strategy for query optimizer hinting. 60
4.5 The local selection strategy for query optimizer hinting for different workloads and systems. 64
a Speedup Stack on PG 12.4. 64
b Speedup Stack on PG 14.6. 64
c Speedup JOB on PG 14.6. 64
4.6 Speedup of local and global models for Stack on PG 12.4. 65
4.7 Speedup runtime model maintenance for Stack on PG 14.6. 66

5.1 The cost model for compression algorithm selection by Damme et al. [25]. . 71
5.2 The global learning selection strategy as a classification problem. 73
5.3 The local learning selection strategy with regression models for one objective. 74
5.4 Different user-data-independent data generator techniques for BWHs. . . . 76
a The la-ola generator. 76
b The outlier generator. 76
c The tidal generator. 76
5.5 The two system setup of our integer selection strategy. 79
5.6 Overall quality of the local selection strategy. 83
a Accuracy (higher is better). 83
b Slowdown (lower is better). 83
5.7 The local selection strategy for compression on different hardware. 85
a Accuracy (higher is better). 85
b Slowdown (lower is better). 85

LIST OF TABLES

3.1 Operator table 31
3.2 Extended operator table 39
3.3 Model complexities for JOB-light. 47
3.4 Q-error for different models and featurizations on JOB-light. 47
3.5 Times of forward passes. 49
3.6 Execution times for training workloads with grouping sets (GS). 50

4.1 End-to-end runtimes and speedups (in brackets) of our local learning strategy for different benchmarks. 63

5.1 Important features for integer compression data. 77
5.2 Performance of neural networks and gradient boosting. 78
5.3 Relative error for objectives (avg. over all algorithms). 81
5.4 Adding new algorithms to the pool only marginally affects performance. . 81

6.1 Comparison of local learning strategies for different applications. 90

A.1 All results from the local selection strategy for integer compression evaluation. 111

A

APPENDIX

Table A.1: All results from the local selection strategy for integer compression evaluation.

generator	data set	objective	accuracy baseline	slowdown baseline	accuracy cost model	slowdown cost model	accuracy local	slowdown local
la-ola	TODS	compr ram2ram	65.81%	26.35%	91.35%	11.00%	69.27%	10.59%
		decompr ram2ram	55.89%	2.07%	20.81%	1.56%	72.22%	1.31%
		compr cache2ram	61.46%	37.44%	68.71%	34.37%	70.16%	8.98%
		decompr ram2reg	53.32%	43.91%	43.17%	32.38%	72.78%	13.96%
		compression rate	37.32%	37.24%	74.70%	35.68%	64.36%	7.21%
	publicBI	compr ram2ram	96.51%	8.69%	96.67%	8.23%	45.93%	9.55%
		decompr ram2ram	45.84%	0.54%	5.83%	1.25%	64.71%	0.36%
		compr cache2ram	92.42%	14.87%	92.69%	14.52%	90.41%	13.41%
		decompr ram2reg	76.43%	9.72%	72.17%	17.09%	62.79%	8.38%
		compression rate	47.97%	14.63%	52.36%	13.08%	51.14%	6.74%
outlier	TODS	compr ram2ram					81.62%	18.88%
		decompr ram2ram					62.21%	2.05%
		compr cache2ram					78.30%	22.86%
		decompr ram2reg					70.86%	25.28%
		compression rate					54.96%	20.57%
	publicBI	compr ram2ram					82.03%	15.11%
		decompr ram2ram					8.19%	1.25%
		compr cache2ram					78.11%	39.91%
		decompr ram2reg					62.79%	8.38%
		compression rate					29.08%	8.85%
tidal	TODS	compr ram2ram					84.61%	9.26%
		decompr ram2ram					80.36%	1.45%
		compr cache2ram					83.72%	19.86%
		decompr ram2reg					85.31%	24.96%
		compression rate					83.54%	21.81%
	publicBI	compr ram2ram					96.53%	8.42%
		decompr ram2ram					61.84%	0.37%
		compr cache2ram					92.61%	14.54%
		decompr ram2reg					82.10%	8.35%
		compression rate					48.93%	11.45%

CONFIRMATION

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

L. Wolf

Dresden, July 30th, 2023