Structure-Aware Machine Learning over Multi-Relational Databases

Maximilian-Joël Schleich
Kellogg College
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy in Computer Science
Trinity 2019

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor Prof. Dan Olteanu for his guidance, invaluable advice, and the many hours we spent discussing the project. His support was a crucial factor of the success of this thesis, and I could not have asked for a better supervisor.

A big, heartfelt thank you goes out to my colleagues at relationalAI, in particular Hung Ngo, Mahmoud Abo Khamis, and Long Nguyen, for their contributions to much of the work presented in this thesis. The thesis would not have been possible without their support. I also thank Molham Aref for providing me with the opportunity to spend one summer at relationalAI in Berkeley.

I appreciate all the support and companionship my colleagues from the Oxford FDB group both in the lab and outside. Their presence has made the past years significantly more enjoyable.

On a more personal note, I owe eternal gratitude to my family, in particular my parents and my brother, for their continuous support and encouragement. I would not have been able to pursue this DPhil without their support.

Lastly and most importantly, thank you Julia, for your support, love, and patience. I look forward to a lifetime of adventures with you.

Abstract

We consider the problem of computing machine learning models over multi-relational databases. The mainstream approach involves a costly repeated loop that data scientists have to deal with on a daily basis: select features from data residing in relational databases using feature extraction queries involving joins, projections, and aggregations; export the training dataset defined by such queries; convert this dataset into the format of an external learning tool; and train the desired model using this tool.

In this thesis, we advocate for an alternative approach that avoids this loops and instead tightly integrates the query and learning tasks into one unified solution. The primary observation is that the data-intensive computation for a variety of learning tasks can be expressed as group-by aggregates over the join of the database relations.

This observation allows us to employ a combination of known and novel state-of-the-art query evaluation techniques, which exploit structure in the query and data to optimize the computation of the aggregates. As a result, we show that, for a class of machine learning models, our integrated, structure-aware approach for the end-to-end learning of models over databases can be asymptotically faster than the mainstream solution that first constructs the feature extraction query. This class of models includes supervised machine learning problems for regression and classification, as well as unsupervised learning problems.

This theoretical development informed the design and implementation of LMFAO (Layered Multiple Functional Aggregate Optimization), an in-memory optimization and execution engine for batches of aggregates over the input database. LMFAO consists of several layers of logical and code optimizations that systematically exploit factorization, sharing of computation, parallelism, and code specialization.

We conducted two types of performance benchmarks. First, we benchmark LMFAO against PostgreSQL, MonetDB, and a commercial database management system for the computation of aggregate batches. Then, we compare the performance of LMFAO against several machine learning packages commonly used in data science for the end-to-end learning pipeline of a variety of models over databases. In all benchmarks, LMFAO is able to outperform the competitors with a speedup of up to three orders of magnitude. In many cases, LMFAO can compute the end-to-end learning pipeline even faster than it takes the machine learning competitors to construct the input training dataset.

Contents

1 Introduction **1**

2 Thesis Contributions **5**
2.1 Methodology for Structure-Aware Machine Learning 5
2.2 System for Structure-Aware Machine Learning 9
2.3 Experimental Evaluation 10

3 Thesis Outline **12**

I Methodology for Structure-Aware Machine Learning **14**

4 Aggregate Queries over Multi-Relational Databases **15**
4.1 Data Model 16
4.2 Query Language 16
4.3 Examples of Aggregate Queries 17
4.4 Feature Extraction Queries 20

5 Complexity of Factorized Aggregate Query Evaluation **21**
5.1 Hypergraphs and Fractional Hypertree Decompositions 22
5.2 Width Measures for Tree Decompositions 25
5.3 Complexity of Aggregate Queries 28
5.4 Evaluating Aggregate Queries with Additive Inequalities over Relaxed Tree Decompositions 30
5.5 Computing Batches of Aggregate Queries 37
5.6 Discussion: Beyond Fractional Hypertree Width 38

6 Optimization Problems with Square Loss Function **42**
6.1 Primer: One-Hot Encoding of Categorical Features 44
6.2 Primer: Batch Gradient Descent Optimization 45
6.3 Linear Regression 46
6.4 Polynomial Regression 52
6.5 Factorization Machines 58
6.6 Model Selection 62
6.7 Alternative Optimization Algorithms 65

7 Optimization Problems with Non-Polynomial Loss Function 69
7.1 Support Vector Machines . . . 70
7.2 Robust Linear Regression . . . 74
7.3 Other Non-Polynomial Loss Functions . . . 78
7.4 Discussion . . . 80

8 Decision Trees 82
8.1 CART Algorithm . . . 83
8.2 Regression Trees . . . 85
8.3 Classification Trees . . . 87

9 Unsupervised Machine Learning 90
9.1 Principal Component Analysis . . . 90
9.2 K-Means Clustering . . . 95
9.2.1 K-Means Clustering via Aggregates with Inequalities . . . 97
9.2.2 K-Means Clustering via Approximation with Coresets . . . 98
9.3 Mutual Information . . . 102
9.4 Data Cubes . . . 104

10 Related Work 107

II The LMFAO System for Structure-Aware Machine Learning 120

11 LMFAO: Layered Multiple Functional Aggregate Optimization 121
11.1 System Overview: The Layers of LMFAO . . . 122

12 View Generation 126
12.1 Join Trees . . . 126
12.2 Computing Aggregates over Join Trees . . . 127
12.3 Find Roots: Each Aggregate to Its Own Root . . . 129
12.4 Aggregate Pushdown over Join Trees . . . 132
12.5 Merging Views . . . 137

13 Multi-Output Optimization 139
13.1 Grouping Views . . . 139
13.2 Join Variable Order . . . 142
13.3 View and Aggregate Registration . . . 144

14 Code Generation 157
14.1 Data Structures . . . 157
14.2 Loop Synthesis . . . 158
14.3 Inlining Function Calls . . . 159
14.4 Parallelization . . . 159

15 Discussion: Applications in LMFAO 160
15.1 Linear Regression . . . 160
15.2 Regression Tree . . . 161
15.3 K-Means . . . 163

16 Related Work **164**

III Experimental Evaluation **167**

17 Experimental Evaluation Summary **168**

18 Datasets **170**

19 Aggregate Computation **172**
19.1 Aggregate Computation Benchmark 172
19.1.1 Competitors 172
19.1.2 Experimental Setup 173
19.1.3 Takeaways 173
19.2 LMFAO Optimization Breakdown 175

20 Learning Predictive Models **177**
20.1 Learning Regression Models 177
20.1.1 Competitors 177
20.1.2 Experimental Setup 178
20.1.3 Takeaways 179
20.2 Learning Decision Trees 181
20.2.1 Experimental Setup 181
20.2.2 Takeaways 182

21 K-Means Clustering **184**
21.1 Comparison with mlpack 184
21.1.1 Experimental Setup. 184
21.1.2 Takeaways 185
21.2 Breakdown of Rk-means 186
21.2.1 Setting $\kappa < k$ for Step 2 187

22 Conclusion and Future Work **188**

Bibliography **189**

Chapter 1

Introduction

Machine learning has the potential to become a general-purpose technology just as computing became general-purpose 70 years ago. Over the past decade, significant progress has been made in the ability to build prediction models, which are the main building blocks in machine learning that turn the data we have into the information we need. This progress can be attributed to three factors: (1) the development of better and more user-friendly machine learning systems, (2) the invention of more performant hardware, and (3) the availability of ever larger amounts of data.

Despite those breakthroughs, the state-of-the-art prediction models remain very expensive to compute. To scale the learning of such models, the common trend in the machine learning systems community is to distribute the computation to many machines and/or GPUs [113, 44, 1]. As a consequence, the full potential of machine learning is confined to tech giants such as Google or Microsoft, which have seemingly unlimited compute farms. In order to extend the benefits of machine learning to wider audiences, it is vital to limit the amount of computation resources that is required to compute the end-to-end pipeline for machine learning workloads.

In this spirit, we put forward an array of optimization techniques, which can make the learning of a wide range of machine learning models feasible on one commodity machine. These optimization techniques are either novel or adaptations of known optimizations that came out of the database theory and systems communities.

According to a 2017 Kaggle survey on the state of data science and machine learning among 16,000 machine learning practitioners [73], the majority of practical data science tasks involve relational data. In retail, insurance, and marketing, more than 80% of used data is relational, in finance it is 77%. This is not surprising. Since its inception in 1969, the relational model has seen a massive adoption in practice. This is due to the development of relational database management systems that scale to large amounts of data, and because the relational model can easily encode domain knowledge. Relational data is rich with knowledge of the underlying domain, which is modeled using database constraints and the result of the investment of many human hours of curation and normalization.

Yet the current state of affairs in building predictive models over relational data largely ignores the structure and rich semantics readily available in relational databases. In fact, the state-of-the-art machine learning systems abstract away from the underlying representation of the data, and require as input a single data matrix.

In the end-to-end pipeline for learning over relational data, the data matrix is first constructed in a data management system, which issues a *feature extraction query* over the database. The materialized data matrix then becomes the input to a statistical learning package which learns the desired model. For the first step, it is common to use open source database management systems, such as PostgreSQL or SparkSQL [139], or query processing libraries, such as Python Pandas [86] and R dplyr [136]. Commonly used statistical learning packages include scikit-learn [106], R [112], TensorFlow [1], and MLlib [88].

The separation of the query processing and learning tasks in the end-to-end learning pipeline has several shortcomings: (1) It may require expensive data export/import, as well as data transformations (e.g., from dense to sparse representations), at the interface of the two systems; (2) the result of the feature extraction query can be much larger than the input database, which causes scalability issues for the learning system; and (3) the data matrix throws away the relational structure that is encoded in the underlying data, which leads to missed optimization opportunities. The combination of these shortcomings stretches the scalability of the learning system. As a result, it is common that data scientists cannot use the full data at their disposal, and learn the model over fewer data sources or a partition of the data provided. By leaving out data sources, however, the model potentially misses important correlations in the data, which leads to less accurate models.

In this thesis, we advocate for an alternative approach to learning over multi-relational databases, which tightly integrates the query processing and learning tasks. This approach is based on the observation that the data-intensive computation for many machine learning problems can be cast into a database problem that involves the computation of batches of aggregate queries over the database. The aggregates compute sufficient statistics required to learn the model, and are much smaller in size than the materialized data matrix.

The coupled evaluation of query processing and learning has several benefits: (1) we can push the computation of aggregate queries past joins in the feature extraction query, and avoid materializing the data matrix; (2) we can exploit the relational structure in the data to optimize the aggregate computation; and (3) we can use recent developments in database query processing, which factorize the computation of aggregate queries to achieve asymptotic runtime improvements over conventional data management systems [10, 101].

Figure 1.1 schematically depicts the workflows of our approach and of the mainstream approach for solving end-to-end machine learning problems. The latter materializes the result of the feature extraction query, exports it out of the database, and imports it as the data matrix in the machine learning tool, where the desired model is learned. We call it *structure-agnostic* since it does not exploit the relational structure of the underly-

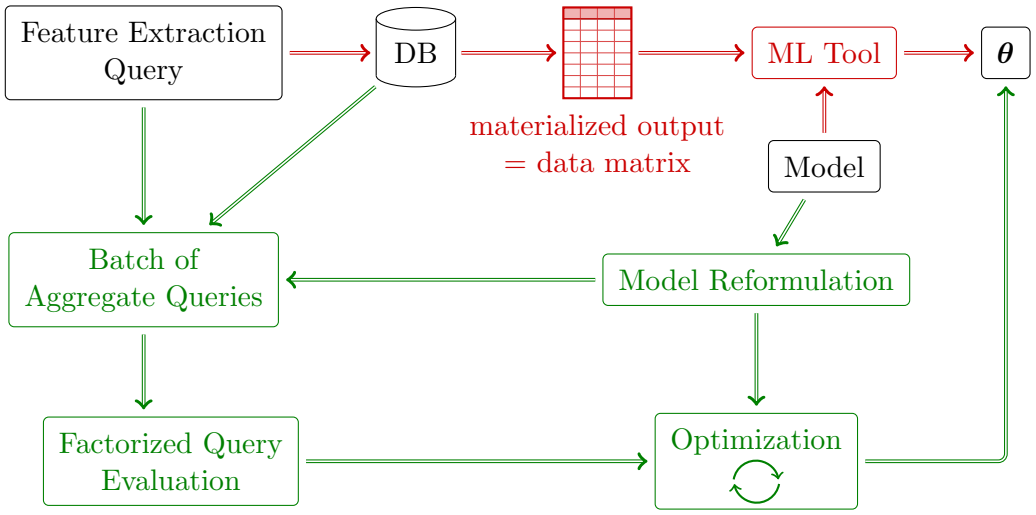


Figure 1.1: **Structure-aware** vs. structure-agnostic learning over relational databases.

ing database, and requires the full materialization of the data matrix. In contrast, our *structure-aware* approach avoids this materialization and has the following steps: (1) it reformulates the data-intensive computation of the model into a batch of aggregates, which form the sufficient statistics needed to solve the optimization problem for the desired model; (2) it computes these aggregates in bulk using factorization techniques and exploiting the relational structure in the data; (3) it solves the learning problem using an optimization algorithm that can be computed directly over the sufficient statistics provided by the computed aggregates; and (4) it returns the model which is equivalent to the one that is learned by the structure-agnostic approach. For some applications, the optimization algorithm may be iterative and require the repeated computation of the aggregate batch over the database.

We identify two challenges for the structure-aware learning of models over databases. The first is the *conceptual challenge*: Which machine learning models can benefit from the tight integration of query processing and learning? In this thesis, we present a large class of models that can benefit from structure-aware learning. This includes supervised machine learning problems for regression (e.g., linear regression, factorization machines, regression trees) and classification (e.g., support vector machines, classification trees), as well as unsupervised learning problems (e.g. k-means clustering, principal component analysis).

In addition to the conceptual challenge, there is the *database systems challenge* which involves the computation of large query batches. Query processing lies at the core of database research, with four decades of innovation and engineering on query engines for relational databases. Without doubt, the efficient computation of a handful of group-by aggregates over a join is well-supported by mature academic and commercial systems and also widely researched. There is, however, relatively less development for large batches of such queries, with initial work in the context of data cubes [59, 65, 91] and SQL-aware data mining systems [32, 33] from two decades ago. In typical structure-aware learning applications, the

number of aggregates in a batch can range from few tens to several thousand, depending on the workload and dataset. Our observation is that the aggregate queries in the batch have very similar structure and are computed over the same join. Therefore, they can benefit from optimizations which systematically share computation across several queries in a batch. In our experimental evaluations, we observe that existing query processing systems fail to exploit these optimization opportunities and thus cannot compute large query batches efficiently. Therefore, the structure-aware learning methods call for new data processing techniques to deal with large query batches.

In this thesis, we address both aforementioned challenges for structure-aware learning. To address the conceptual challenge, we present the methodology for structure-aware learning, and provide runtime guarantees for the end-to-end learning over databases. To address the database systems challenge, we report on the design and implementation of a system, which is designed to compute large batches of aggregate queries and support the efficient computation of structure-aware learning problems. We further present an extensive experimental evaluation of the structure-aware learning system. We benchmark the system against state-of-the-art database management systems for the computation of aggregate batches, as well as several machine learning packages that are commonly used in data science. The experiments show that the system consistently outperforms the competitors, with speedups up to three orders of magnitude.

The contributions of this thesis have been implemented at relationalAI (`https://www.relational.ai/`), and now form the basis of the analytics performed in their integrated engine for database and analytics workloads. We next overview the contributions of this thesis in more detail.

Chapter 2

Thesis Contributions

In this chapter, we provide a summary of the main contributions in this thesis. The contributions are categorized into three parts: (1) We present the methodology and theoretical foundation of learning over databases; (2) we overview LMFAO, a system designed to support structure-aware learning; and (3) we conduct an experimental evaluation that benchmarks LMFAO against state-of-the-art data management and machine learning systems.

2.1 Methodology for Structure-Aware Machine Learning

Part I of the thesis presents the methodology as well as theoretical foundation of structure-aware machine learning over multi-relational databases. This contribution has two components: (1) we show for a variety of learning problems that we can rewrite the data-intensive computation into a batch of aggregate queries; and (2) based on these rewritings, we can then give runtime guarantees for the end-to-end learning over databases. Theses guarantees can be asymptotically faster than the mainstream approach that first materializes the feature extraction query and then solves the learning problem over the result.

Rewriting of data-intensive computation into aggregate queries

The rewriting of the data-intensive computation of the learning problem into aggregate queries exploits the algebraic structure of the problem. In particular, it exploits the distributivity of products over summations to identify subproblems that can be cast into an aggregate query. The aggregate queries in the batch are all computed over the feature extraction query that defines the data matrix for the learning problem. As a result, all queries share the same join body, which provides ample opportunities to share computation. The model learned over the results of the aggregate batch is syntactically equivalent to the model learned over the materialized feature extraction query.

We show the rewriting for the following four classes of models:

(1) First, we consider *supervised machine learning problems* where the objective function

is defined by the *square loss function*. This class of problems includes (ridge and lasso) linear regression, polynomial regression, and factorization machines.

We show that the data-intensive computation of these problems can be computed as large batches of group-by aggregate queries. Once the aggregates are computed, the model can then be learned directly over the aggregate batch using batch-gradient descent optimization. Thus, we are able to decouple the aggregate computation from the learning of the model, and the model optimization is performed independently of the input database. The details are provided in Chapter 6.

We first presented the rewriting for linear regression models with continuous variables [119], and then extended it to polynomial regression models [101]. Subsequently, we presented a similar rewriting for factorization machines, and also introduced a sparse, relational encoding to compute models with categorical variables under one-hot encoding [7]. This publication also presents an approach to exploit functional dependencies (FDs) in the feature space to simplify the computation of the model. In particular, we use FDs to reparameterize polynomial regression models and factorization machines, and to learn a simpler, equivalent model instead. This contribution is not presented in this thesis.

(2) We then consider a more general class of supervised machine learning problems, where the objective function is defined by *non-polynomial loss functions*. This class of problems includes the learning of robust regression models (with Huber loss) and support vector machines (with hinge loss).

In contrast to (1), the queries that capture the data-intensive computation for this class of problems are subject to *additive inequalities*, which form selection conditions on the query result. In addition, when learned with (sub-)gradient based optimization algorithms, we cannot fully decouple the learning task from the aggregate computation. Instead, we require that the aggregates are recomputed for each optimization step. Nevertheless, we show that the learning can be asymptotically faster than first materializing the feature extraction query.

Chapter 7 presents the details on the rewritings for a range of non-polynomial loss functions. We first presented these contributions in [5].

(3) We further consider the learning of *decision trees* for both regression and classification. We show that the data-intensive computation for the learning of decision trees can be directly cast into simple group-by aggregate queries (without additive inequalities).

Decision trees also belong to the class of supervised learning problems. We separate decision trees from the other two classes, because they require a different optimization algorithm. Whereas classes (1) and (2) can be optimized with a global optimization algorithm (e.g., gradient descent), decision trees are learned with a greedy procedure,

which learns the tree one node at a time. The learning algorithm requires that the aggregate batch is recomputed for each decision tree node. Nevertheless, we can compute the entire decision tree asymptotically faster than the time it takes to materialize the feature extraction query.

Chapter 8 presents the details on the aggregate batch that forms the data-intensive computation for learning decision trees. We previously presented this result in [118].

(4) Finally, we consider unsupervised machine learning problems, including the computation of principal component analysis, k-means clustering, mutual information between two discrete random variables, and data cubes. This is covered in Chapter 9.

For principal component analysis, we show that the structure-aware approach requires the computation of a similar aggregate batch as our solution to square-loss problems. In particular, we can compute the covariance matrix of the data matrix D, which is used to compute the principal components, without materializing the feature extraction query that represents D. We first presented this contribution in [8].

For the k-means clustering, we show how the data-intensive computation of the algorithm can be reformulated into aggregate queries with additive inequalities. We first presented this reformulation in [5].

The data-intensive computation for the mutual information of two discrete variables involves computing simple group-by aggregate queries (without additive inequalities). We further present three common applications of mutual information: (1) feature selection, (2) learning the structure of Bayesian networks via the Chow-Liu algorithm, and (3) learning decision trees. A subset of the results were previously shown in [118].

Finally, computing data cubes is a classic problem in data mining, and requires the computation of many group-by aggregate queries, which are related to the aggregates required by our square-loss solution. We first presented the connection between data cubes and structure-aware learning in [118].

Runtime bounds for end-to-end pipeline

In order to characterize the runtime complexity of the end-to-end pipeline for learning over databases, we rely on recent developments in the database theory community, in particular factorized query evaluation algorithms [10, 101].

In a nutshell, factorized query evaluation algorithms exploit the structure of the query and the data to avoid redundant computation for the evaluation of a single aggregate. They achieve this by unifying three powerful ideas: worst-case optimal join processing, query plans defined by fractional hypertree decompositions of join queries, and pushing aggregates past joins. It has been shown that the runtime of the algorithms is proportional to fractional hypertree width (fhtw), which is a width measure used to capture tractability

of a host of problems across Computer Science, spanning from databases to logic, CSP, SAT, and matrix computation [10]. The factorized evaluation of aggregate queries can be asymptotically faster than materializing the feature extraction query.

In this thesis, we further present an novel approach for the evaluation of aggregate queries with additive inequalities. This approach couples known factorized evaluation algorithms with geometric data structures. This extension allows us to relax conditions for the hypertree decomposition that define the query plans, and to define a new width measure, called the *relaxed* fractional hypertree width (r-fhtw). Using this approach, the overall runtime for the evaluation of aggregate queries with additive inequalities can be faster by a polynomial factor than existing approaches. We first presented this insight in [5].

The publication also presents a novel algorithm for the evaluation of aggregate queries, called #PANDA [5]. In contrast to the factorized evaluation algorithms considered in this thesis, #PANDA decomposes the query into several subproblems and then solves each subproblem with the respective optimal tree decomposition. The runtime of #PANDA is proportional to a novel width measure, called the *sharp submodular width* (#subw). For queries with additive inequalites, we also present the *relaxed* #subw, which is analogous to r-fhtw. We prove that #subw $\leq$ fhtw, and there are queries for which #subw is unboundedly smaller than fhtw. Chapter 5.6 presents a high-level overview of these results, and also discusses why we do not use this algorithm in the context of this thesis. For a more detailed overview, we refer the reader to the original publication [5].

The rewriting of the data-intensive computation into aggregate queries, as well as the runtime bounds for factorized query evaluation algorithms, allows us to characterize the end-to-end runtime for learning over databases. We show that, for a large class of feature extraction queries, it is possible to compute the entire end-to-end learning pipeline for a model *asymptotically faster* than the time it takes to materialize the feature extraction query. This makes our structure-aware approach competitive *regardless of the learning techniques* used by the mainstream approaches whenever they first materialize the training dataset. This includes sampling techniques that are commonly used in machine learning to speed-up the computation of the model. We previously presented the bounds in [7, 8, 5].

The methodology for structure-aware learning over databases presented in this thesis is the first to connect tractability measures developed by the theoretical computer science and algorithms communities to learning models, thereby establishing the computational complexity of learning models over relational databases.

Beyond relational structure

There are learning problems for which the rewriting of the data-intensive computation into aggregate queries does not result in asymptotic runtime improvements. In such cases, we conjecture that we can nevertheless improve the end-to-end runtime with our structure-aware approach, if we exploit not only the relational structure in the data and query, but

also the geometric and statistical structure of the learning problem.

To support this conjecture, we present a novel clustering algorithm, called Rk-means, which exploits the geometric structure of the k-means clustering objective to compute the clusters over a small coreset instead of the full result of the feature extraction query. The coreset guarantees that the final result is a constant factor approximation of the k-means objective, and can be computed as a batch of aggregate queries (without additive inequalities). Thus, we are able to compute a good approximation to the objective, while benefiting from the asymptotic runtime improvements brought by factorized query evaluation.

The details on the Rk-means algorithm are presented in Section 9.2.2. We first presented the algorithm in [42], which also provides the proofs for the constant factor approximation guarantees of the coreset.

2.2 System for Structure-Aware Machine Learning

Part II presents LMFAO (Layered Multiple Functional Aggregate Optimization), an in-memory execution engine for batches of aggregates over relational data, and the main systems contribution of this thesis.

LMFAO is outcome of several attempts of designing systems for structure-aware learning. In particular, LMFAO generalizes and extends on two predecessors: (1) F [119], which supports linear regression models, and (2) AC/DC [6], which generalizes F to polynomial regression models and factorization machines, as well as the efficient support for categorical variables. In contrast to F and AC/DC, LMFAO supports arbitrary group-by aggregate queries, and extends the class of supported learning problems. This class includes learning decision trees for regression and classification, computing mutual information, Chow-Liu trees, data cubes, and k-means clustering with the Rk-means algorithm. We do not report on the design of F and AC/DC for reasons of space.

The main motivation for LMFAO stems from two observations.

The first observation is that our structure-aware approach to learning over databases requires the computation of very large batches of aggregate queries. In our experimental evaluation, the number of aggregates in a batch can range from few tens to several thousand, depending on the workload and dataset.

The second observation follows from our experimental evaluation, which shows that state-of-the-art data management systems fail to compute large batches of queries efficiently. In our evaluation with PostgreSQL, MonetDB, and a commercial system, we observe that these systems are not able to exploit the fact that all queries in the batch have similar structure. They are able to compute each individual query efficiently, but fail to share computation across queries. In some cases, the workload also reaches internal design limitations, e.g., the maximum number of columns in PostgreSQL is much less than the number of model features.

To compute batches of aggregates efficiently, LMFAO employs several layers of optimization techniques. The optimizations are either novel or adaptations of known concepts to our specific workload, and systematically exploit factorization, sharing of computation, parallelism, and code specialization. Next, we briefly overview the high-level optimization layers of LMFAO. The full summary and the algorithms for the optimization layers of LMFAO are presented in Part II. We first presented the key-design choices of LMFAO in [118].

LMFAO takes as input the batch of aggregate queries, the database schema, and cardinality constraints, such as sizes of relations and variable domains. The *View Generation* layer then decomposes each query in the batch into views over a join tree. The optimizations in this layer are concerned with logical transformations of view expressions.

The *Multi-Output Optimization* layer constructs groups of views, where each view group can be computed over the join of a single relation and incoming views. The layer then constructs an optimized evaluation plan for each view group, which computes all views in the group in one pass over the underlying join. Since this plan outputs the results for several views, we call it the *multi-output execution plan*. We perform several optimizations on this execution plan, which factorize the computation of the aggregates, and share the computation across views in the group. The optimizations at this layer cannot be expressed at the syntactic level of the query language.

Finally, the *Code Generation* layer compiles the multi-output evaluation plan from the previous layer into efficient and specialized C++ code. The components of this layer perform several low-level code optimizations, which include optimizing cache locality, inlining function calls, and parallelizing the computation.

2.3 Experimental Evaluation

Part III of the thesis presents the experimental evaluation of LMFAO. We conduct two performance benchmarks in LMFAO. First, we benchmark the computation of batches of aggregates in LMFAO and state-of-the-art data management systems. Then, we compare the end-to-end performance of learning models over databases in LMFAO against machine learning libraries that are commonly used in practical data science applications.

Aggregate computation

We show that by designing for the workload required by learning tasks, LMFAO can outperform general-purpose mature database systems such as PostgreSQL, MonetDB, and a commercial database system by orders of magnitude. This is not only a matter of query optimization, but also of execution. Aspects of LMFAO's optimized execution for query batches can be cast into SQL and fed to a database system. Such SQL queries capture decomposition of aggregates into components that can be pushed past joins and shared across aggregates, and as such they may create additional intermediate aggregates. This poses

scalability problems to these systems due to, e.g., design limitations such as the maximum number of columns or lack of efficient query batch processing, and leads to larger compute times than for the plain unoptimized queries. This hints at LMFAO's distinct design that departs from mainstream query processing.

End-to-end pipeline for learning over databases

The performance advantage brought by LMFAO's design becomes even more apparent for the end-to-end applications. We compare LMFAO against state-of-the-art mainstream structure-agnostic solutions, e.g., TensorFlow [1], R [112], Scikit-learn [106], mlpack [41] and MADlib [67]. For four real dataset from the retail and advertising domains, the competitors either take orders of magnitude more time than LMFAO to learn the same model or fail due to various design limitations. In many cases, LMFAO is able to compute the end-to-end solution for the learning problem faster that it takes PostgreSQL or Python Pandas [86] to materialize the data matrix.

In this thesis, we present the results for the learning of linear regression models, regression trees, and classification trees. A subset of these results were previously presented in [118]. In a previous publication [6], we present similar results for the learning of polynomial regression models and factorization machines, and investigate the effect of reparameterizing the model using functional dependencies.

In addition, we present an experimental evaluation of the Rk-means algorithm for clustering over small coresets, where the coreset is computed in LMFAO. The results show that Rk-means can solve the clustering problem orders of magnitude faster than mlpack [41] with only moderate approximation errors. The results are consistent with the approximation guarantees of the algorithm, and were previously presented in [42].

This performance improvement is due to the difference in the size of the input data for the learning algorithm. LMFAO computes aggregate batches over the input database whose size ranges from tens of KBs to hundreds of MBs. The learning algorithm computed over the aggregate batch takes relatively insignificant time. In contrast, the competitor systems require the materialization of the feature extraction query. For our datasets, the resulting data matrix have tens of GBs, and can be an order of magnitude larger than the input databases used to create them.

In addition to being expected to work on much larger inputs, the machine learning libraries are less scalable than the data systems. These solutions also inherit the limitations of both of their underlying systems, e.g., the maximum data frame size in R and the maximum number of columns in PostgreSQL are much less than typical database sizes and respectively number of model features.

The experimental evaluation thus supports a main conjecture of this thesis: scalability challenges of state-of-the-art machine learning systems can be mitigated by a combination of database systems techniques.

Chapter 3

Thesis Outline

This thesis is structured into three parts, following the strands of contributions outlined above. Part I presents the methodology and theoretical foundation of structure-aware learning, and is composed of the following chapters.

- Chapter 4 introduces and exemplifies a query language that is used to express the aggregate queries. The language can express (1) the aggregate queries that form the sufficient statistics for structure aware learning, and (2) feature extraction queries, which define the input data matrix for structure-agnostic machine learning models.
- Chapter 5 provides runtime bounds for the evaluation of aggregate queries that can be expressed in the language presented in Chapter 4. In particular, we introduce novel runtime bounds for the evaluation of aggregate queries with additive inequalities.
- Chapter 6 provides our structure-aware solution for supervised machine learning problems, where the objective function is defined by the square loss. This class of problems contains linear regression, polynomial regression, and factorization machines.
- Chapter 7 considers supervised machine learning problems where the objective function is defined by a non-polynomial loss function. This class of problems includes robust regression models and support vector machines.
- Chapter 8 presents our structure-aware solution to learning decision trees. We consider both regression and classification trees.
- Chapter 9 shows our structure-aware solution for the computation of principal component analysis, k-means clustering, mutual information between two discreet random variables, and data cubes.
- Chapter 10 presents related work, and overviews the current landscape systems for learning over relational databases.

For each model discussed in Chapters 6 to 9, we present the runtime for the end-to-end structure-aware learning of the model and show that the problem can be solved asymptotically faster than the time it takes to materialize the data matrix.

Part II presents the key design choices of the LMFAO system for structure-aware learning, and consists of the following chapters.

- Chapter 11 introduces LMFAO.
- Chapter 11.1 presents a high-level summary of the system.
- Chapters 12, 13, and 14 expand on the key design choices for each layer and optimization step behind LMFAO.
- Chapter 15 discusses how LMFAO computes and optimizes linear regression models, decision trees, and k-means clustering.
- Chapter 16 presents related work for the optimizations performed by LMFAO, and compares them with optimizations commonly employed in machine learning systems.

Part III presents the experimental evaluation of LMFAO, and consists of the following chapters:

- Chapter 17 presents an overview of the reported benchmarks.
- Chapter 18 provides a summary of the datasets used for the experiments.
- Chapter 19 presents an evaluation of the computation of aggregate batches in LMFAO, PostgreSQL, MonetDB, and a commercial data management system.
- Chapter 20 presents benchmarks for the learning of linear regression models, regression trees, and classification trees over databases in LMFAO, TensorFlow, scikit-learn and MADlib.
- Chapter 21 presents an experimental evaluation of k-means clustering in LMFAO and mlpack.

Finally, Chapter 22 concludes the thesis and presents additional challenges for future work.

Part I

Methodology for Structure-Aware Machine Learning

Chapter 4

Aggregate Queries over Multi-Relational Databases

In this chapter, we consider aggregate queries that are evaluated over multi-relational databases. Aggregate queries are at the core of the solutions presented in this thesis, for two reasons: (1) We use aggregate queries to capture the data intensive computation for the optimization of machine learning models, and (2) they also capture the class of feature extraction queries, which define the training dataset for models that are learned over multi-relational databases.

We express these queries in a query language that is inspired by Datalog, and more concise than an equivalent formulation in SQL or functional aggregate queries over a single semiring (FAQ-SS) [10]. We exemplify several queries in this language, which capture various stages in the end-to-end pipeline for learning models over databases. We also show that queries expressed in this language can be translated into FAQ-SS [10]. Finally, we introduce feature extraction queries.

Chapter Outline

The outline of this chapter is as follows:

- Section 4.1 presents the data model used in this thesis.
- Section 4.2 defines the query language that we use to define aggregate queries.
- Section 4.3 exemplifies several queries in the query language, which capture several stages in the end-to-end pipeline for learning models over multi-relational databases.
- Section 4.4 introduces feature extraction queries.

4.1 Data Model

Throughout the thesis, we use the following convention. Uppercase X_i denotes a variable, and lowercase x_i denotes a value in the domain $\mathsf{Dom}(X_i)$ of variable X_i. For any positive integer n, let $[n]$ denote the set $\{1, \ldots, n\}$.

A schema ω is a tuple of variables, but, for simplicity, we allow for set operations of the schema. A tuple $\boldsymbol{x}_\omega$ of data values over schema ω is a type-consistent mapping from ω to domain $\mathsf{Dom}(\omega) = \prod_{X \in \omega} \mathsf{Dom}(X)$. A relation R with schema ω_R is a finite set of tuples over ω_R. The size $|R|$ of R is the number of its tuples. A database I is a set of relations. We consider the RAM model of computation.

4.2 Query Language

Consider a relational database I with m relations $\{R_j(\omega_{R_j})\}_{j \in [m]}$. We consider the problem of evaluating group-by aggregate queries over the join of the relations in I.

To simplify notation, we assume that each relation symbol occurs only once. To capture queries with self-joins, we assume without loss of generality that mappings of relation symbols to database relations, as well as a correspondence between the schemas of relation symbols and database relations, are given together with the query. We thus consider queries with self-joins, though avoid the explicit use of aliases and renaming operators in the algebraic expressions.

We define aggregate queries over the join of relations $\{R_j(\omega_{R_j})\}_{j \in [m]}$ as follows.

Definition 4.1. An aggregate query Q over database I with relations $R_1, \ldots, R_m$ can be expressed as:

$$Q(\omega_Q, \texttt{SUM(}\alpha_1\texttt{)}, \ldots, \texttt{SUM(}\alpha_\ell\texttt{)}) \leftarrow R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}), \phi_1(\omega_{\phi_1}), \ldots, \phi_t(\omega_{\phi_t}), \tag{4.1}$$

where:

- For any two relations R_i and R_j, the set $\omega_{R_i} \cap \omega_{R_j}$ are the join variables between R_i and R_j. The conjunction $R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m})$ defines the natural join of the relations in I.
- $\mathcal{V} = \bigcup_{i \in [m]} \omega_{R_i}$ is the set of variables of Q. Overloading notation, we often use $\mathcal{V} = [n]$ as the index set of the variables $\{X_i\}_{i \in [n]}$ that occur in query Q.
- $\omega_Q \subseteq \mathcal{V}$ is the set of group-by variables of Q, which are also called *free* variables.
- For each $i \in [t]$, ϕ_i with schema $\omega_{\phi_i} \subseteq \mathcal{V}$ encodes an additive inequality over $|\omega_{\phi_i}|$ (uni-variate) functions $\gamma_v^{\phi_i} : \mathsf{Dom}(X_v) \to \mathbb{R}$:

$$\phi_i(\omega_{\phi_i}) = \sum_{v \in \omega_{\phi_i}} \gamma_v^{\phi_i}(X_v) \leq 0. \tag{4.2}$$

The functions $\gamma_v^{\phi_i}$ can be user-defined functions (e.g., $\gamma_1^{\phi_i}(X_1) = X_1^2/2$), or binary predicates with one key in $\mathsf{Dom}(X_v)$ and a numeric value. The only requirement is that, given x, the value $\gamma_v^{\phi}(x)$ can be accessed/computed in $\tilde{O}(1)$-time.

The inequalities $\{\phi_i\}_{i\in[t]}$ encode selection conditions on the tuples in the join of the relations $\{R_i\}_{i\in[m]}$.

- For each $i \in [\ell]$, α_i is a sum of products of user-defined aggregate functions (UDAF) $f_{ijk} : \mathsf{Dom}(\omega_{f_{ijk}}) \to \mathbb{R}$ with $j \in [s_i]$ and $k \in [p_{ij}]$ where $s_i, p_{ij} \in \mathbb{N}$:

$$\alpha_i = \sum_{j\in[s_i]} \prod_{k\in[p_{ij}]} f_{ijk}(\omega_{f_{ijk}}). \tag{4.3}$$

  We assume that each function $f_{ijk}(\omega_{f_{ijk}})$ can be evaluated in $\tilde{O}(1)$-time.

- The integers ℓ and t can be zero, in which case the query does not contain an aggregate and respectively an inequality. □

The query formulation also supports `MIN` and `MAX` aggregates, by replacing `SUM` with the respective operator. For this thesis, we focus on `SUM` aggregates as it is the most common aggregate used to capture the data-intensive computation of machine learning problems.

As in SQL, we assume bag semantics for the relations. An equivalent (albeit more verbose) SQL formulation for Q is:

```
CREATE VIEW Q AS
SELECT   ω_Q, SUM(α_1), ..., SUM(α_ℓ)
FROM     R_1 NATURAL JOIN R_2 ... NATURAL JOIN R_m
WHERE    φ_1(ω_φ1) AND φ_2(ω_φ2) ... AND φ_t(ω_φt)
GROUP BY ω_Q;
```

Our queries are FAQ-SS [10] queries that allow for multiple aggregates with arbitrary UDAFs. Aggregate queries with additive inequalities are also called FAQ-AI queries [5].

4.3 Examples of Aggregate Queries

We next provide several examples for queries of the form (4.1), which are used in the following chapters to express computations for learning models.

Count and Sum Aggregates. Consider the following query Q with group-by variables ω_Q that is computed over the join of relations $R_1, \ldots, R_m$:

$$Q(\omega_Q, \mathtt{SUM}(1)) \leftarrow R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}).$$

The query Q is of the form (4.1), where α encodes a single constant function $f(\,) = 1$. It computes the number of tuples for each group over variables ω_Q in the join result.

Similarly, the following query Q computes the sum over variable $X_i \in \mathcal{V}$ for each group over variables ω_Q in the join of the relations $R_1, \ldots, R_m$:

$$Q(\omega_Q, \texttt{SUM}(X_i)) \leftarrow R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}).$$

This query is of the form (4.1), where α encodes the unary function $f(X_i) = X_i$.

Conjunctive Queries. A conjunctive query Q is defined as follows:

$$Q(\omega_Q) \leftarrow R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}). \tag{4.4}$$

The query Q projects the join result onto the variable set $\omega_Q \subseteq \mathcal{V}$. It is a query of the form (4.1) that does not compute an aggregate. If $\omega_Q = \mathcal{V}$, query Q is called a join query.

Learning Linear Regression Models. Queries of the form (4.1) can also compute the data-intensive computation for a range of machine learning problems. As an example, we consider the case of learning a linear regression model over a training dataset D with variables $\boldsymbol{X} = (X_1, \ldots, X_n)$ and label X_{n+1}. We assume that this dataset is the result of a natural join query Q over relations $R_1, \ldots, R_m$.

Given a vector $\boldsymbol{x} = (x_i)_{i \in [n]}$ over (continuous) variables $\boldsymbol{X}$, a linear regression model with parameters $\boldsymbol{\theta} = (\theta_i)_{i \in [n]}$ is defined as follows:

$$\mathsf{LR}(\boldsymbol{x}) = \sum_{j \in [n]} \theta_j \cdot x_j. \tag{4.5}$$

The parameters of a linear regression model can be learned by minimizing an objective function $J(\boldsymbol{\theta})$ with the square-loss function:

$$J(\boldsymbol{\theta}) = \frac{1}{2|D|} \sum_{(\boldsymbol{x}, x_{n+1}) \in D} \big(\sum_{j \in [n]} \theta_j \cdot x_j - x_{n+1} \big)^2. \tag{4.6}$$

A popular algorithm to minimize $J(\boldsymbol{\theta})$ is the batch gradient descent (BGD) optimization algorithm. The details of the BGD algorithm are presented in Section 6.2. For this example we focus on the data-intensive computation: For each $k \in [n]$, BGD repeatedly updates parameter $\boldsymbol{\theta}_k$ in the direction of the partial derivative of the objective with respect to θ_k:

$$\frac{\partial}{\partial \boldsymbol{\theta}_k} J(\boldsymbol{\theta}) = \frac{1}{|D|} \sum_{\boldsymbol{x} \in D} \left(\sum_{j \in [n]} \theta_j \cdot x_j - x_{n+1} \right) x_k. \tag{4.7}$$

The partial derivative of $J(\boldsymbol{\theta})$ can be expressed as the following aggregate query:

$$Q_k(\texttt{SUM}(\sum_{i\in[n]}(\theta_i \cdot X_i \cdot X_k) - X_{n+1} \cdot X_k) \leftarrow R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}).$$

The query Q_k is a query of the form (4.1) with one aggregate, where $\alpha = \sum_{i\in[n+1]} f_i(X_i) \cdot g_k(X_k)$ is a summation over a products of the functions $g_i(X_i) = X_i$ and $f_i(X_i) = \theta_i \cdot X_i$, $\forall i \in [n]$. For the case where $i = n+1$, we set $f_{n+1}(X_{n+1}) = -X_{n+1}$.

K-means Clustering. The k-means clustering algorithm divides the input dataset D into k clusters of "similar" data points $D = \bigcup_{i\in[k]} D_i$ where the similarity is defined with respect to the Euclidean distance and k is a given fixed positive integer [72]. Each cluster D_i is represented by a cluster mean $\boldsymbol{\mu}_i \in \mathbb{R}^n$. The k-means problem is to find the cluster means $(\boldsymbol{\mu}_i)_{i\in[k]}$ that minimize the optimization problem:

$$\min_{(\boldsymbol{\mu}_1,\ldots,\boldsymbol{\mu}_k)} \sum_{\boldsymbol{x}\in D} \min_{i\in[k]} \left(\|\boldsymbol{x} - \boldsymbol{\mu}_i\|_2^2 \right).$$

We present algorithms to solve this optimization problem in Section 9.2. For this example, we focus on the data-intensive computation of the problem: computing the mean vectors $(\boldsymbol{\mu}_i)_{i\in[k]}$ for the k clusters.

Let $\mu_{i,\ell}$ denote the ℓ'th component of $\boldsymbol{\mu}_i$. For a data point $\boldsymbol{x} \in D$, the function c_{ij} computes the difference between the squares of the ℓ_2-distances from $\boldsymbol{x}$ and $\boldsymbol{\mu}_i$ and from $\boldsymbol{x}$ and $\boldsymbol{\mu}_j$:

$$c_{ij}(\mathcal{V}) = \sum_{\ell\in[n]} [\mu_{i,\ell}^2 - 2 \cdot X_\ell \cdot (\mu_{i,\ell} + \mu_{j,\ell}) - \mu_{j,\ell}^2].$$

A data point $\boldsymbol{x} \in D$ is closest to mean $\boldsymbol{\mu}_i$ from the set of k kmeans if and only if $\forall j \in [k] : c_{ij} \leq 0$. For simplicity, we use $\phi_{ij}(\mathcal{V}) = c_{ij} \leq 0$ to encode this inequality.

To compute the mean vector $\boldsymbol{\mu}_i$, we need to compute the sum of values for each dimension $\ell \in [n]$ over D_i. If the dataset D is the join of database relations $(R_j)_{j\in[m]}$ over schemas ω_{R_j}, we can formulate this sum computation as the following aggregate query:

$$Q_i\,(\texttt{SUM(1)}, \texttt{SUM}(X_1), \ldots, \texttt{SUM}(X_n)) \leftarrow R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}), \phi_{i1}(\mathcal{V}), \ldots, \phi_{ik}(\mathcal{V}).$$

This query computes $n+1$ SUM-aggregates over all tuples in the join result that are closer to the point $\boldsymbol{\mu}_i$ than any other $(\boldsymbol{\mu}_\ell)_{\ell\in[k],\ell\neq i}$. It is precisely of the form (4.1) with k additive inequalities.

4.4 Feature Extraction Queries

Most machine learning algorithms require as input a single data matrix D. This data matrix may represent the training dataset for machine learning models, the input for multi-dimensional data cubes, or the joint probability distribution to be approximated by a Bayesian network. For learning over multi-relational databases, the data matrix D is defined by a *feature extraction query* over the database.

A feature extraction query Q is typically a conjunctive (or join) query of the form (4.4), where the variables in the query define features for the machine learning model. The query is computed over the join of the relations in the input database. It is also common to join in interpreted relations that are derived from the input relations by aggregating some of their columns. These further relations provide derived features, which are added to the raw features readily provided by the input relations. For ease of notation, we assume that the input and interpreted relations are materialized. We further assume that the query is provided as input and designed by a domain expert. In the following, we use $\boldsymbol{X}$ to denote the variables in the query result, such that a tuple $\boldsymbol{x} \in D$ over the variables $\boldsymbol{X}$ defines a feature vector for the model. In supervised learning scenarios, the query Q also returns a label Y, which defines the quantity that the model aims to predict.

Example 4.2. Consider the following join query Q that is a simplified version of a feature extraction query for a retail forecasting scenario:

$$
\begin{aligned}
&Q(\mathsf{sku}, \mathsf{store}, \mathsf{color}, \mathsf{quarter}, \mathsf{city}, \mathsf{country}, \mathsf{units\ sold}, \mathsf{price}, \mathsf{size}) \\
&\quad \leftarrow \mathsf{Sales}(\mathsf{sku}, \mathsf{store}, \mathsf{day}, \mathsf{units\ sold}), \mathsf{Items}(\mathsf{sku}, \mathsf{color}, \mathsf{price}), \mathsf{Quarter}(\mathsf{day}, \mathsf{quarter}), \\
&\qquad \mathsf{Stores}(\mathsf{store}, \mathsf{city}, \mathsf{size}), \mathsf{Country}(\mathsf{city}, \mathsf{country}).
\end{aligned}
$$

Relation Sales records the number of units of a given sku (stock keeping unit) sold at a store on a particular day. The retailer is a global business, so it has stores in different cities and countries. One objective is to predict the number of blue units to be sold next year in the Fall quarter in Berlin. The label is the variable units sold, and $\mathcal{V} = \{\mathsf{sku}, \mathsf{store}, \mathsf{day}, \mathsf{color}, \mathsf{quarter}, \mathsf{city}, \mathsf{country}, \mathsf{units\ sold}, \mathsf{price}, \mathsf{size}\}$ is the set of all variables. All variables except units sold and day define features in the model, i.e., $\boldsymbol{X} = \{\mathsf{sku}, \mathsf{store}, \mathsf{color}, \mathsf{quarter}, \mathsf{city}, \mathsf{country}, \mathsf{pricex}, \mathsf{size}\}$.

In many practical applications, the query Q is extended with additional aggregates. Common examples of such aggregations include: the number of sales in the past week or month, the average sales per city, the number of days since the item has been on promotion. It is also common to use unary transformations for features, e.g $\log(\mathsf{price})$, which are used to model non-linear dependencies between the feature and the label. □

Chapter 5

Complexity of Factorized Aggregate Query Evaluation

In this chapter, we present runtime bounds for the evaluation of aggregate queries of the form (4.1). We first recall known runtime bounds that follow from recent advances in algorithms for query evaluation [10, 101]. We then show that for aggregate queries with additive inequalities we can improve the runtime bound of these algorithms, by coupling them with geometric data structures. The analysis presented in this chapter forms the foundation for the theoretical analysis of the solutions for machine learning problems that are presented in the following chapters.

At the core of the complexity results is the observation that relational query processing has a high degree of redundancy in both representation and computation. This redundancy is not necessary for subsequent computation, including the learning of models over query results. Instead, it is possible to factorize both the computation and the representation of the query result, which can lead to exponential runtime improvements in combined complexity. This observation is the foundation for the factorized databases (FDB) [101] and functional aggregate query (FAQ) [10] frameworks, which provide efficient algorithms for factorized aggregate query evaluation.

In a nutshell, the algorithms used by FDB and FAQ for factorized query processing unify three powerful ideas: worst-case optimal join processing, query plans defined by fractional hypertree decompositions of join queries, and pushing aggregates past joins. In this chapter, we present the runtime bounds for these algorithms.

For the evaluation of aggregate queries with additive inequalities, we define *relaxed* fractional hypertree decompositions, and show that by extending the factorized query processing algorithm with Chazelle's geometric data structures [34], we can evaluate the queries over such decompositions with strictly lower runtime than FDB and FAQ. In addition, we present a lower bound that shows that this result is the best we can hope for.

Following the introduction of the runtime bounds for aggregate queries, we prove a result

that is central to the analysis in the following chapters. We consider batches of aggregate queries which are computed over the same join of relations in a database. We show that for infinitely many join queries, we can compute the aggregate batch asymptotically faster than the join result.

The results for aggregate queries with additive inequalities have been presented in [5]. The background information on factorized aggregate computation is based on [102, 101, 10].

Chapter Outline

The outline for this chapter is as follows:

- Section 5.1 introduces query hypergraphs and fractional hypertree decomposition of hypergraphs. We also introduce *relaxed* fractional hypertree decompositions.
- Section 5.2 then introduces common width measures of fractional hypertree decompositions, which are used to characterize the complexity of evaluating aggregate queries.
- Section 5.3 then provides runtime bounds for computing aggregate queries of the form (4.1), which follows from the factorized query evaluation algorithms of the FDB and FAQ frameworks.
- Section 5.4 considers aggregate queries with additive inequalities, and shows that we can extend the algorithms presented in Section 5.3 with Chazelle's geometric data structures for semigroup range searching to achieve a better runtime bound for the evaluation of the query.
- Section 5.5 proves the result that batches of aggregate queries over joins can be computed sub-linear in the time it takes to materialize the join result.
- Finally, Section 5.6 discusses recent results in database query processing, which show that aggregate queries of the form 4.1 can be computed with a better runtime bound if the query plan uses multiple fractional hypertree decompositions.

5.1 Hypergraphs and Fractional Hypertree Decompositions

In this section, we introduce hypergraphs for queries of the form (4.1) and fractional hypertree decompositions of hypergraphs. A hypergraph is a generalization of an undirected graph where the hyperedges of the hypergraph are defined as arbitrary nonempty subsets of the vertex set instead of only strictly size 2 subsets.

We use $\mathcal{H} = (\mathcal{V}, \mathcal{E} = \mathcal{E}_R \cup \mathcal{E}_\infty)$ to denote the hypergraph of a query Q of the form (4.1), where the vertices are given by the set $\mathcal{V}$ of variables in Q. The set $\mathcal{E}$ of hyperedges is partitioned in two sets $\mathcal{E}_R$ and $\mathcal{E}_\infty$. The hyperedge set $\mathcal{E}_R$ consists of one hyperedge for the variable set ω_S of each relation S in Q; and $\mathcal{E}_\infty$ has one hyperedge per set ω_ϕ for each

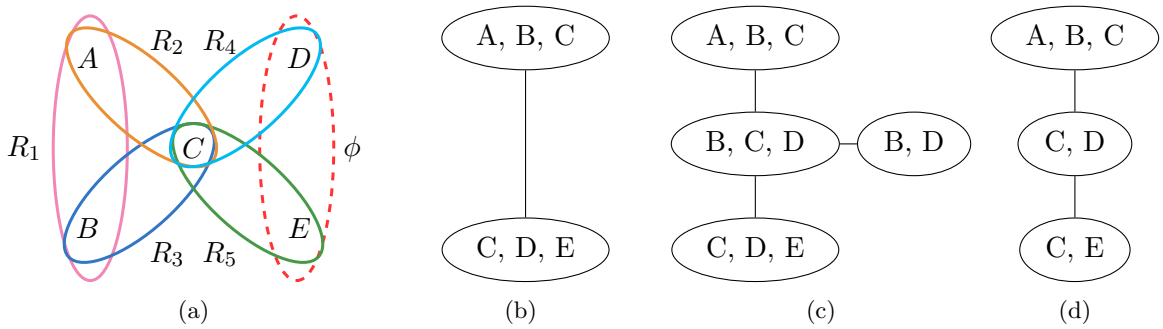


Figure 5.1: (a) The hypergraph $\mathcal{H}$ of query Q from Example 5.1 (annotated by the relation name), (b) a tree decompositions of $\mathcal{H}$, (c) a $\{B, D\}$-connex tree decomposition, and (d) a relaxed tree decomposition of $\mathcal{H}$, where ϕ is covered by the bottom two adjacent bags.

additive inequality ϕ in the body of Q. We use ∞ as index for the set of additive inequality hyperedges to specify that additive inequalities may have infinite support, i.e. $|\phi| = \infty$.

Example 5.1. Consider the following query:

$$Q(A, B, C, D, E) \leftarrow R_1(A, B), R_2(A, C), R_3(B, C), R_4(C, D), R_5(C, E), \phi(D, E)$$

with the inequality $\phi(D, E) = D + E < 0$.

The query hypergraph of Q is $\mathcal{H} = (\mathcal{V}, \mathcal{E} = \mathcal{E}_R \cup \mathcal{E}_\infty)$, where $\mathcal{V} = \{A, B, C, D, E\}$, $\mathcal{E}_R = \{R_1(A, B), R_2(A, C), R_3(B, C), R_4(C, D), R_5(C, E)\}$, and $\mathcal{E}_\infty = \{\phi(D, E)\}$. Figure 5.1(a) depicts the hypergraph $\mathcal{H}$, where the inequality edge for ϕ is shown dashed and in red. □

We next introduce fractional hypertree decompositions of query hypergraphs and the fractional hypertree width. Decompositions of the query hypergraph measure the "degree of acyclicity" of the query. They are traditionally used for classifying the tractability of Boolean queries and constraint satisfaction problems. We refer the reader to the recent survey by Gottlob et al. [56] for more details and historical context.

Definition 5.2 (Fractional hypertree decomposition [56])**.** Let $\mathcal{H} = (\mathcal{V}, \mathcal{E} = \mathcal{E}_R \cup \mathcal{E}_\infty)$ be a hypergraph. A *fractional hypertree decomposition* of $\mathcal{H}$ is a pair (T, χ) where $T = (V(T), E(T))$ is a tree and $\chi : V(T) \to 2^{\mathcal{V}}$ assigns to each node of the tree T a subset of vertices of $\mathcal{H}$. The sets $(\chi(t))_{t \in V(T)}$, are called the *bags* of the tree decomposition. There are two properties the bags must satisfy:

(a) *Coverage property:* For any hyperedge $S \in \mathcal{E}$, there is a bag $(\chi(t))_{t \in V(T)}$, such that $S \subseteq \chi(t)$.

(b) *Running intersection property:* For any vertex $v \in \mathcal{V}$, the set $\{t \mid t \in V(T), v \in \chi(t)\}$ is not empty and forms a connected subtree of T. □

We use $\mathsf{TD}(\mathcal{H})$ denote the set of all fractional hypertree decompositions of $\mathcal{H}$. In the following, we refer to fractional hypertree decompositions simply as tree decompositions.

Definition 5.3 (F-connex tree decomposition [22, 121])**.** Given a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ and a set $F \subseteq \mathcal{V}$, a tree decomposition (T, χ) of $\mathcal{H}$ is *F-connex* if $F = \emptyset$ or the following holds: There is a nonempty subset $V' \subseteq V(T)$ that forms a connected subtree of T and satisfies $\bigcup_{t \in V'} \chi(t) = F$. □

Let $\mathsf{TD}_F(\mathcal{H})$ denote the set of all F-connex tree decompositions of $\mathcal{H}$. Note that whenever $F = \emptyset$ or $F = \mathcal{V}$, the set $\mathsf{TD}_F(\mathcal{H})$ of all F-connex tree decompositions is equivalent to the set $\mathsf{TD}(\mathcal{H})$ of all tree decompositions of $\mathcal{H}$.

Definition 5.4 (Acyclicity)**.** A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ without additive inequalities is *α-acyclic* if and only if there exists a tree decomposition (T, χ) in which every bag $\chi(t)$ is a hyperedge of $\mathcal{H}$. □

When $\mathcal{H}$ represents a join query, the tree T in the above definition is also called the *join tree* of the query. A query is acyclic if and only if its hypergraph is α-acyclic.

We next introduce a novel notion of tree decomposition, called *relaxed* tree decomposition, which relaxes the coverage property for hyperedges that define an additive inequality.

Definition 5.5 (Relaxed tree decomposition [5])**.** Let $\mathcal{H} = (\mathcal{V}, \mathcal{E} = \mathcal{E}_R \cup \mathcal{E}_\infty)$ denote a hypergraph whose hyperedge set is partitioned into $\mathcal{E}_R$ and $\mathcal{E}_\infty$. A *relaxed* tree decomposition of $\mathcal{H}$ is a pair (T, χ), where $T = (V(T), E(T))$ is a tree, and $\chi : V(T) \to 2^{\mathcal{V}}$ satisfies the following properties:

(a) *Running intersection property:* For each node $v \in \mathcal{V}$ the set $\{t \in V(T) \mid v \in \chi(t)\}$ is a connected subtree in T.

(b) Every "relation" hyperedge $S \in \mathcal{E}_R$ there is a bag $(\chi(t))_{t \in V(T)}$, such that $S \subseteq \chi(t)$.

(c) Every "inequality" hyperedge $S \in \mathcal{E}_\infty$ there are two *adjacent* bags $\chi(s)$ and $\chi(t)$, such that: $S \subseteq \chi(s) \cup \chi(t)$. □

Definition 5.6 (F-connex relaxed tree decomposition [5])**.** For a set $F \subseteq \mathcal{V}$, a *relaxed* tree decomposition (T, χ) of $\mathcal{H}$ is *F-connex* if $F = \emptyset$ or the following holds: There is a nonempty subset $V' \subseteq V(T)$ that forms a connected subtree of T and satisfies $\bigcup_{t \in V'} \chi(t) = F$. □

Let $\mathsf{TD}^{\mathsf{r}}(\mathcal{H})$ and $\mathsf{TD}^{\mathsf{r}}_F(\mathcal{H})$ denote the set of all relaxed tree decompositions and respectively relaxed F-connex tree decompositions of $\mathcal{H}$.

Example 5.7. Figure 5.1(b) shows a valid tree decomposition for the query from Example 5.1. Each bag in the tree decomposition covers one of the two triangles in the hypergraph, thus satisfying the coverage property. This tree decomposition is also F-connex, whenever F is $\{A, B, C\}$, $\{C, D, E\}$, or $\{A, B, C, D, E\}$.

If $F = \{B, D\}$, then a valid F-connex tree decomposition is shown in Figure 5.1(c). Note that the bag $\{B, D\}$ is contained in $\{B, C, D\}$, in which case the latter typically "absorbs" the former, so that the tree decomposition consists of only three bags.

The inequality ϕ encodes an additive inequality of the form (4.2). A relaxed tree decomposition relaxes the coverage property for the hyperedge $\{D, E\}$, so that this edge only needs to be covered by two adjacent bags in the tree decomposition. A corresponding relaxed tree decomposition is shown in Figure 5.1(d). □

5.2 Width Measures for Tree Decompositions

In this section, we introduce a range of width measures for tree decompositions of query hypergraphs, which are based on the fractional edge cover number of the hypergraph.

Definition 5.8 (Fractional edge cover number ρ^*)**.** Let $\mathcal{H} = (\mathcal{V}, \mathcal{E} = \mathcal{E}_R \cup \mathcal{E}_\infty)$ be a hypergraph (of some query Q of the form (4.1)). Let $B \subseteq \mathcal{V}$ be any subset of the vertices in $\mathcal{H}$. A *fractional edge cover* of B using edges $\mathcal{E}_R$ in $\mathcal{H}$ is a feasible solution $\boldsymbol{\lambda} = (\lambda_S)_{S \in \mathcal{E}_R}$ to the following linear program:

$$\begin{aligned} \min & \sum_{S \in \mathcal{E}_R} \lambda_S \\ \text{s.t.} & \sum_{S: v \in S} \lambda_S \geq 1, \quad \forall v \in B \\ & \lambda_S \geq 0, \quad \forall S \in \mathcal{E}_R. \end{aligned}$$

The optimal objective value of the above linear program is called the *fractional edge cover number* of B in $\mathcal{H}$ and is denoted by $\rho^*_{\mathcal{H}}(B)$. When $\mathcal{H}$ is clear from the context, we drop the subscript $\mathcal{H}$ and use $\rho^*(B)$.

The fractional edge cover number of a query Q with hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E} = \mathcal{E}_R \cup \mathcal{E}_\infty)$ is $\rho^*_{\mathcal{H}}(\mathcal{V})$. □

By Definition 5.8, one does not assign a weight to hyperedges in $\mathcal{E}_\infty$.

Example 5.9. Consider the query Q with hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E} = \mathcal{E}_R \cup \mathcal{E}_\infty)$ from Example 5.1. The fractional edge cover number of Q is $\rho^*_{\mathcal{H}}(\mathcal{V}) = 3$. This is because the variables D and E only occur in relations R_4 and respectively R_5. Thus, to cover D and E, we need to assign a weight of 1 to the hyperedges that R_4 and R_5. This assignment also covers variable C. The remaining variables A and B can be covered by assigning a weight of 1 to the hyperedge for relation R_1. The hyperedges R_2 and R_3 are assigned a weight of 0. The sum over the weights of all hyperedges in $\mathcal{E}_R$ is 3. One can verify that no other valid assignment of weights to hyperedges can give a smaller weight sum, and thus $\rho^*_{\mathcal{H}}(\mathcal{V}) = 3$. □

We next define the fractional hypertree width for hypergraphs. The fractional hypertree width is fundamental to problem tractability with applications spanning constraint satisfaction, databases, matrix operations, logic, and probabilistic graphical models [10].

Definition 5.10 (Fractional hypertree width [84])**.** Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a hypergraph. The *fractional hypertree width* of a tree decomposition (T, χ) of $\mathcal{H}$ is

$$\mathsf{fhtw}((T, \chi)) = \max_{t \in V(T)} \rho^*_{\mathcal{H}}(\chi(t)). \tag{5.1}$$

The *fractional hypertree width of* $\mathcal{H}$ is the *minimum* fractional hypertree width over all tree decompositions of $\mathcal{H}$:

$$\mathsf{fhtw}(\mathcal{H}) = \min_{(T,\chi) \in \mathsf{TD}(\mathcal{H})} \mathsf{fhtw}((T, \chi)). \tag{5.2}$$

The *F-connex fractional hypertree width of* $\mathcal{H}$ is the *minimum* fractional hypertree width of all F-connex tree decompositions of $\mathcal{H}$:

$$\mathsf{fhtw}_F(\mathcal{H}) = \min_{(T,\chi) \in \mathsf{TD}_F(\mathcal{H})} \mathsf{fhtw}((T, \chi)). \tag{5.3}$$

□

The F-connex fractional hypertree width is a generalization of the fractional hypertree width from Boolean to arbitrary conjunctive queries. It is equivalent to the factorization width that was denoted by $s^{\uparrow}(Q)$ in [102], and generalized to FAQ-width to capture the complexity of Functional Aggregate Queries over several semirings [10]. Note that $\mathsf{fhtw}_F(\mathcal{H}) = \mathsf{fhtw}(\mathcal{H})$ when $F = \emptyset$ or $F = \mathcal{V}$.

We next define the *relaxed* variant of fractional hypertree width.

Definition 5.11 (Relaxed Fractional Hypertree Width)**.** Let $\mathcal{H} = (\mathcal{V}, \mathcal{E} = \mathcal{E}_R \cup \mathcal{E}_\infty)$ denote a hypergraph whose hyperedge set is partitioned into $\mathcal{E}_R$ and $\mathcal{E}_\infty$.

The *relaxed fractional hypertree width of* $\mathcal{H}$ is the *minimum* fractional hypertree width over all *relaxed* tree decompositions of $\mathcal{H}$:

$$\mathsf{r\text{-}fhtw}(\mathcal{H}) = \min_{(T,\chi) \in \mathsf{TD}^{\mathsf{r}}(\mathcal{H})} \mathsf{fhtw}((T, \chi)). \tag{5.4}$$

For $F \subseteq \mathcal{V}$, the *relaxed* F-connex fractional hypertree width of $\mathcal{H}$ is the *minimum* fractional hypertree width over all *relaxed* F-connex tree decompositions of $\mathcal{H}$:

$$\mathsf{r\text{-}fhtw}_F(\mathcal{H}) = \min_{(T,\chi) \in \mathsf{TD}^{\mathsf{r}}_F(\mathcal{H})} \mathsf{fhtw}((T, \chi)). \tag{5.5}$$

□

The relaxed fractional hypertree width was generalized as the relaxed FAQ-width in [5].

Example 5.12. The tree decomposition in Figure 5.1(b) has $\mathsf{fhtw} = 2$. The explanation is as follows. The variables D and E only occur in relations R_4 and respectively R_5. So in order to cover D and E, we need to assign a weight of 1 to the hyperedges that define R_4 and R_5. Therefore, the bag $\{C, D, E\}$ has $\rho^*(\{C, D, E\}) = 2$. For the bag $\{A, B, C\}$, we can cover the

variable A, B, and C by assigning a weight of $\frac{1}{2}$ to the hyperedges of the relations R_1, R_2, and R_3, which gives $\rho^*(\{A, B, C\}) = \frac{3}{2}$. Thus, $\mathsf{fhtw} = \max(\rho^*(\{A, B, C\}), \rho^*(\{C, D, E\})) = 2$.

The tree decomposition in Figure 5.1(c) also has $\mathsf{fhtw} = 2$. This follows from the explanation above and the fact that the bags $\{B, C, D\}$, and $\{B, D\}$ are covered by the hyperedges for relations R_3 and R_4.

The *relaxed* tree decomposition in Figure 5.1(d) has $\mathsf{r\text{-}fhtw} = \frac{3}{2}$, which is the fractional edge cover number of the triangle that defines the bag $\{A, B, C\}$. The other two bags both have $\rho^* = 1$. Therefore, the relaxation of the coverage property can lower the fhtw. □

For a given query Q with query hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E} = \mathcal{E}_R \cup \mathcal{E}_\infty)$, it is known that

$$1 \leq \mathsf{fhtw}(\mathcal{H}) \leq \rho^*_{\mathcal{H}}(\mathcal{V}) \leq |\mathcal{E}_R|,$$

where the gap between $\mathsf{fhtw}(\mathcal{H})$ and $\rho^*(\mathcal{H})$ can be as large as $|\mathcal{E}_R|$ [102]. Furthermore, fhtw can be less than other width notions such as (generalized) hypertree width and tree width [58]. We next give a bound for the F-connex fractional hypertree width.

Proposition 5.13 (Adopted from [8])**.** *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a hypergraph of a query of the form* (4.1) *and $F \subseteq \mathcal{V}$. When $|F| \geq 1$, we have*

$$\mathsf{fhtw}(\mathcal{H}) \leq \mathsf{fhtw}_F(\mathcal{H}) \leq \mathsf{fhtw}(\mathcal{H}) + |F| - 1.$$

Proof. The lower bound $\mathsf{fhtw}(\mathcal{H}) \leq \mathsf{fhtw}_F(\mathcal{H})$ follows from the definition of fhtw and the fact that $\mathsf{TD}_F(\mathcal{H}) \subseteq \mathsf{TD}(\mathcal{H})$. We next proof the upper bound $\mathsf{fhtw}_F(\mathcal{H}) \leq \mathsf{fhtw}(\mathcal{H}) + |F| - 1$.

Find a tree decomposition (T, χ) of $\mathcal{H}$ with minimal fhtw, i.e. where $\mathsf{fhtw}((T, \chi)) = \mathsf{fhtw}(\mathcal{H})$. Without loss of generality let the free variables be $F = \{v_1, \ldots, v_f\}$. Construct another tree decomposition $(T, \overline{\chi})$ by extending all bags $\chi(t)$ of (T, χ) with the variables $\{v_2, \ldots, v_f\}$, i.e. by defining $\overline{\chi}(t) = \chi(t) \cup \{v_2, \ldots, v_f\}$ for all t. By Definition 5.2, $(T, \overline{\chi})$ *is* a tree decomposition of $\mathcal{H}$. Since $\rho^*\left(\chi(t) \cup \{v_2, \ldots, v_f\}\right) \leq \rho^*(\chi(t)) + f - 1$, we have

$$\mathsf{fhtw}((T, \overline{\chi})) \leq \mathsf{fhtw}((T, \chi)) + |F| - 1.$$

Moreover, since (T, χ) must have a bag $\chi(t^*)$ that contains v_1, the corresponding bag $\overline{\chi}(t^*)$ of $(T, \overline{\chi})$ contains all the free variables $\{v_1, \ldots, v_f\}$. If $\overline{\chi}(t^*)$ has additional non-free variables, we can add a new bag to $(T, \overline{\chi})$ which is adjacent to $\overline{\chi}(t^*)$ and only contains the free variables $\{v_1, \ldots, v_f\}$. The fractional edge cover number of the new bag cannot be higher than $\rho^*(\overline{\chi}(t^*))$. □

Similarly, we can give a bound for the *relaxed* fractional hypertree width.

Proposition 5.14 (Adopted from [5])**.** *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a hypergraph of a query of the*

form (4.1) *and* $F \subseteq \mathcal{V}$. *Then, we have*

$$\frac{1}{2}\mathsf{fhtw}_F(\mathcal{H}) \leq \mathsf{r\text{-}fhtw}_F(\mathcal{H}) \leq \mathsf{fhtw}_F(\mathcal{H}).$$

Proof. Since $\mathsf{TD}_F(\mathcal{H}) \subseteq \mathsf{TD}^{\mathsf{r}}_F(\mathcal{H})$, by definition $\mathsf{r\text{-}fhtw}_F(\mathcal{H})$ cannot be larger than $\mathsf{fhtw}_F(\mathcal{H})$. We next proof the lower bound $\frac{1}{2}\mathsf{fhtw}_F(\mathcal{H}) \leq \mathsf{r\text{-}fhtw}_F(\mathcal{H})$.

Let (T, χ) denote a relaxed tree decomposition of $\mathcal{H}$ with fractional hypertree width $\mathsf{r\text{-}fhtw}(\mathcal{H})$. Construct a new (non-relaxed) tree decomposition (T', χ') for $\mathcal{H}$ as follows. Each vertex t in $V(T)$ is also a vertex in $V(T')$ with $\chi'(t) = \chi(t)$. Moreover, to each edge $\{s, t\} \in E(T)$ there corresponds an additional vertex st in $V(T')$ whose bag is $\chi'(st) = \chi(s) \cup \chi(t)$. We now consider the edge set of T'. For each edge $\{s, t\} \in E(T)$, there are two corresponding edges in $E(T')$, namely $\{s, st\}$ and $\{t, st\}$. It is easy to see that (T', χ') is a (non-relaxed) tree decomposition of $\mathcal{H}$ with width at most $2\mathsf{fhtw}(\mathcal{H})$. Moreover, if (T, χ) is F-connex, then so is (T', χ'). □

With the above definitions, we can now characterize the runtime complexity for evaluating single aggregate queries of the form (4.1).

5.3 Complexity of Aggregate Queries

We consider an aggregate query Q of the form (4.1), which computes a single aggregate over the join of relations $(R_j)_{j\in[t]}$ and inequalities $(\phi_j)_{j\in[t]}$:

$$Q(\omega_Q, \texttt{SUM(}\alpha\texttt{)}) \leftarrow R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}), \phi_1(\omega_{\phi_1}), \ldots, \phi_t(\omega_{\phi_t}), \tag{5.6}$$

where $\omega_Q \subseteq \mathcal{V}$ denotes the free variables of Q and the aggregate expression α encodes a sum of s products of user-defined aggregate functions:

$$\alpha = \sum_{j\in[s]} \prod_{k\in[p_j]} f_{jk}(\omega_{f_{jk}}), \text{ where } s, p_j \in \mathbb{N}.$$

In the following, we assume that the query plan for Q is defined by a tree decomposition (T, χ) of the query hypergraph, and impose the following restriction on the functions in α: for each function $f_{ijk}(\omega_{f_{ijk}})$, there exists a node $t \in V(T)$ with bag $\chi(t)$ in the tree decomposition such that $\omega_{f_{ijk}} \subseteq \chi(t)$, i.e. function f_{ijk} is covered by the bag $\chi(t)$. For a given tuple $\boldsymbol{x}_{\omega_{f_{ij}}}$, by assumption each function f_{jk} can be evaluated in $\tilde{O}(1)$ time.

A naïve evaluation strategy for Q would first materalize the join and then compute the aggregate. This is highly inefficient and involves a lot of redundant computation. An alternative strategy is to factorize the computation of the aggregate, which is done by the FDB [101] and FAQ [10] frameworks. The algorithms for factorized aggregate computation push aggregates past joins, and then eliminate one bag at a time in a bottom-up pass

over a tree decomposition using worst-case optimal join algorithms [133, 96, 97]. In the following, we focus on the runtime complexity of these algorithms which forms the basis of the theoretical analysis in this thesis.

Theorem 5.15 ([101, 10])**.** *Let Q denote an aggregate query of the form* (5.6) *that is computed over a database I, and $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ denote the hypergraph of Q. The size of the query result is denoted by $|Q|$, and the size of each relation in I is at most N. Then, Q can be computed in time*

$$O\left(|\mathcal{V}|^2 \cdot |\mathcal{E}| \cdot s \cdot \left(N^{\mathsf{fhtw}_{\omega_Q}(\mathcal{H})} + |Q|\right) \cdot \log N\right), \tag{5.7}$$

where s is the number of products in the aggregate expression α of Q. □

The proof of Theorem 5.15 then follows directly from the runtime complexity of the algorithms for factorized aggregate comptutation provided by the FDB and FAQ frameworks [10, 102], and the fact that each UDAF in α is covered by some bag in the tree decomposition and can be evaluated in $\tilde{O}(1)$. If the aggregate expression α is a summation of products of UDAFs (i.e., $s > 1$), then this is equivalent to evaluating s independent queries and then summing over their results. This explains the additional s factor in the complexity statement.

For the case that Q does not have group-by variables, i.e., $\omega_Q = \emptyset$, the query Q computes a single aggregate and $|Q| = 1$. The query can be computed in time $O(|\mathcal{V}|^2 \cdot |\mathcal{E}| \cdot N^{\mathsf{fhtw}(\mathcal{H})} \cdot \log N)$. Recall that if Q does not compute an aggregate, then it denotes a conjunctive query. Thus, Theorem 5.15 also gives a bound on the computation time for conjunctive queries Q with free variables ω_Q.

If query Q has $\ell > 1$ aggregate expressions, then we can, without loss of generality, transform this query in ℓ single aggregate queries of the form (5.6) where the runtime of each such query follows from Theorem 5.15.

Size bounds for the Result of Q

In Theorem 5.15, the factor $|Q|$ denotes the size of the output, which depends on the representation of the query result. The conventional relational representation of the result of Q is a listing of the tuples over the free variables ω_Q and the value for the aggregate. We refer to this representation as the listing representation hereafter. Under this representation, the size of the query result can be bounded by $|Q| = O(N^{\rho^*_{\mathcal{H}}(\omega_Q)})$, where $\mathcal{H}$ is the hypergraph of Q. This result follows from the seminal AGM bound [19, 60] for full conjunctive queries. Note that $O(N^{\rho^*_{\mathcal{H}}(\omega_Q)})$ can be larger than $O(N^{\mathsf{fhtw}_F(\mathcal{H})})$, which means that the materialization of the output can become the dominant factor in the complexity of Theorem 5.15. In particular, for a full conjunctive query Q, we can bound the output size of the listing representation by $O(N^{\rho^*_{\mathcal{H}}(\mathcal{V})})$.

The listing representation of query results entails a high degree of redundancy, which is can be avoided by using a *factorized representation* of the query result instead. A factorized representation, or *factorization* for short, exploits the laws of relational algebra, in particular the distribution of the Cartesian product over union that underlies algebraic factorization, to give a succinct and lossless representation of the query result [101]. In addition, factorized representations of query results allow for constant-delay enumeration, which means the time delay between the enumeration from one tuple to the next depends at most on the size of the query but not the size of input.

For a query Q with free variables ω_Q, a valid factorized representation of the query result is given by the ω_Q-connex subtree in the ω_Q-connex tree decomposition that is used to evaluate the query. Instead of materializing the listing representation of the bags in the F-connex subtree, the evaluation algorithm instead represents the result as a collection of bags, where all other bags in the decomposition were aggregated away all other bags.

We can compute the factorized representation of the query result in the time given by Theorem 5.15, where the size of the query result is $|Q| = O(N^{\mathsf{fhtw}_{\omega_Q}(\mathcal{H})})$ [102, 101]. Thus, under the factorized representation of Q, $|Q|$ is always upper bounded by the time it takes to compute the aggregates, and thus we can avoid the term $|Q|$ in the complexity statement in Theorem 5.15.

5.4 Evaluating Aggregate Queries with Additive Inequalities over Relaxed Tree Decompositions

In this section, we show that Q can be computed more efficiently over *relaxed* tree decompositions, which require that any hyperedge in $\mathcal{E}_\infty$ is covered by two adjacent bags. We exemplify the high level idea with a simple example.

Example 5.16. Consider the following `COUNT` query over two binary relations with an inequality between them:

$$Q(\texttt{SUM(1)}) = R(A,B), S(B,C), A \leq C \tag{5.8}$$

The query hypergraph $\mathcal{H}$ of Q has $\mathcal{V} = \{A, B, C\}$, $\mathcal{E}_R = \{\{A, B\}, \{B, C\}\}$, and $\mathcal{E}_\infty = \{\{A, C\}\}$. Together the hyperedges in $\mathcal{E}_R$ and $\mathcal{E}_\infty$ form a triangle with one edge between any two vertices in $\mathcal{V}$.

The algorithms for factorized aggregate computation from Section 5.3 would compute this query over a tree decomposition which has one bag that contains $\mathcal{V}$. This bag has $\rho^*_{\mathcal{H}}(\{A, B, C\}) = 2$, and thus $\mathsf{fhtw}_F(Q) = 2$. Intuitively, the evaluation strategy of the algorithm would first compute the join of R and S, and then count the number of tuples that satisfy the inequality. If $|R| = |S| = N$, this evaluation strategy takes $\tilde{O}(N^2)$ time.

An alternative evaluation strategy avoids materializing the join and instead precomputes

an index for S, which, for every tuple $(b, c) \in S$, returns $|\sigma_{B=b \wedge C \geq c} S|$, i.e., the number of tuples $(b', c') \in S$, for which $b = b'$ and $c \leq c'$. Assuming the relation S is sorted, then this index can be computed in one bottom up pass over the relation.

To compute Q, the algorithm takes each tuple $(a, b) \in R$, does a binary look up on the index of S to find the first tuple (b, c) for which $a \leq c$, and adds the precomputed count in the index to the output. This evaluation strategy takes $O(N \cdot \log N)$ time. □

The problem of evaluating additive inequalities between two bags in a tree decomposition can be reduced to dominance range counting, and the index from Example 5.16 is generalized by Chazelle's classic geometric data structures for solving semigroup range search problems.

In the following, we give a short introduction to semigroup range searching and geometric data structures. We than show the connection between geometric data structures and simplified queries of the form (4.1), which have only two relations (or bags in a tree decomposition). Finally, we show that an extension of the algorithms for factorized aggregate computation using Chazelle's geometric data structures can answer queries of the form (4.1) over relaxed tree decompositions in time given by the relaxed fractional hypertree width.

5.4.1 Semigroup Range Searching

Orthogonal range counting (and searching) is a classic and ubiquitous problem in computational geometry [43]: given a set S of N points in a d-dimensional space, build a data structure that, given any d-dimensional rectangle, can efficiently return the number of enclosed points. More generally, there is the *semigroup range searching* problem [34], where each point $\boldsymbol{p} \in S$ of the N input points also has a weight $w(\boldsymbol{p}) \in G$, where $(G, \oplus)$ is a *semigroup*.[1] The problem is: given a d-dimensional rectangle R, compute $\bigoplus_{\boldsymbol{p} \in S \cap R} w(\boldsymbol{p})$.

Classic results by Chazelle [34] show that there are data structures for *semigroup range searching* which can be constructed in time $O(N \log^{d-1} N)$, and answer rectangular queries in $O(\log^{d-1} N)$-time. This is almost the best we can hope for [35]. There are more recent improvements to Chazelle's result (see, e.g., Chan et al. [31]), but they are minor (at most a log factor), as the original results were already very close to matching the lower bound.

Most of these range search/counting problems can be reduced to the *dominance range searching* problem (on semigroups), where the query is represented by a point $\boldsymbol{q}$, and the objective is to return $\bigoplus_{\boldsymbol{q} \preceq \boldsymbol{p}, \boldsymbol{p} \in S} w(\boldsymbol{p})$. Here, $\preceq$ denotes the "dominance" relation (coordinate-wise $\leq$). We can think of $\boldsymbol{q}$ as the lower-corner of an infinite rectangle query.

[1] In a semigroup we can add two elements using $\oplus$, but there is no additive inverse.

5.4.2 Connecting Aggregate Queries and Geometric Data Structures

To show the connection between geometric data structures and queries of the form (5.6), we start with a special case of (5.6) in which the body only has two relations U and L:

$$Q(\omega_Q, \texttt{SUM(1)}) = U(\omega_U), L(\omega_L), \sum_{v \in \omega_{\phi_1}} \gamma_v^{\phi_1}(X_v) \leq 0, \ldots, \sum_{v \in \omega_{\phi_t}} \gamma_v^{\phi_t}(X_v) \leq 0, \tag{5.9}$$

where U and L are two input relations over schemas ω_U and ω_L, respectively. We prove the following simple but important lemma:

Lemma 5.17. *Let $N = \max\{|U|, |L|\}$, and $k = \max(|\mathcal{E}_\infty| - 1, 1)$, then when $\omega_Q \subseteq \omega_U$, query (5.9) can be answered in time $O(N \cdot \log^k N)$.*

Proof. Without loss of generality, we can assume that, $S \not\subseteq \omega_L$ and $S \not\subseteq \omega_U$ for all $S \in \mathcal{E}_\infty$. If there is a hyperedge $S \in \mathcal{E}_\infty$ for which $S \subseteq \omega_U$, then in a $O(N \log N)$-time pre-processing step we can "absorb" the inequality $\sum_{v \in S} \gamma_v^S(x_v) \leq 0$ into the relation U, by replacing $U(\omega_U)$ with its partition that satisfies the inequality:

$$U(\omega_U) \leftarrow U(\omega_U), \sum_{v \in S} \gamma_v^S(X_v) \leq 0.$$

A similar absorption can be done with $S \subseteq L$.

Abusing notation somewhat, for each $S \in \mathcal{E}_\infty$ and each $T \subseteq S$, define the function $\gamma_T^S : \prod_{v \in T} \mathsf{Dom}(X_v) \to \mathbb{R}$ by

$$\gamma_T^S(\boldsymbol{x}_T) := \sum_{v \in T} \gamma_v^S(X_v). \tag{5.10}$$

To show the connection between query (5.9) and geometric data structures, we translate (5.9) into an equivalent functional aggregate query with additive inequalities [10, 5].

We encode the relation U as a function $\Phi_U : \prod_{X_v \in \omega_U} \mathsf{Dom}(X_v) \to \{0, 1\}$. For a given tuple $\boldsymbol{x}_{\omega_U}$ over the schema ω_U of relation U, the function Φ_U can be expressed with the Kronecker delta:

$$\Phi_U(\boldsymbol{x}_{\omega_U}) = \mathbf{1}_{\boldsymbol{x}_{\omega_U} \in U}.$$

In other words, $\Phi_U(\boldsymbol{x}_{\omega_U}) = 1$, iff $\boldsymbol{x}_{\omega_U} \in U$, and 0 otherwise. Similarly, we encode the relation L as a function Φ_L.

In addition, we encode each inequality $\sum_{v \in S} \gamma_v^S(X_v) \leq 0$ as a function $\varphi_S : \prod_{v \in S} \mathsf{Dom}(X_v) \to \{0, 1\}$, which can also be encoded with the Kronecker delta. For a given tuple $\boldsymbol{x}_S$ over the variables S:

$$\varphi_S(\boldsymbol{x}_S) = \mathbf{1}_{\sum_{v \in S} \gamma_v^S(X_v) \leq 0}.$$

Under this functional encoding, the following functional aggregate query over the sum-

product semiring $(\mathbb{R}, +, \times)$ is an equivalent formulation for (5.9):

$$Q(\boldsymbol{x}_{\omega_Q}) = \sum_{\boldsymbol{x}_{(\omega_U \cup \omega_L)\setminus\omega_Q}} \Phi_U(\boldsymbol{x}_{\omega_U}) \cdot \Phi_L(\boldsymbol{x}_{\omega_L}) \cdot \left(\prod_{S \in \mathcal{E}_\infty} \mathbf{1}_{\sum_{v\in S} \gamma_v^S(x_v) \le 0} \right) \tag{5.11}$$

$$= \sum_{\boldsymbol{x}_{\omega_U \setminus \omega_Q}} \sum_{\boldsymbol{x}_{\omega_L \setminus \omega_U}} \Phi_U(\boldsymbol{x}_{\omega_U}) \cdot \Phi_L(\boldsymbol{x}_{\omega_L}) \cdot \left(\prod_{S \in \mathcal{E}_\infty} \mathbf{1}_{\sum_{v\in S} \gamma_v^S(x_v) \le 0} \right), \tag{5.12}$$

where the summation $\sum_{\boldsymbol{x}_\omega}$ is over tuples $\boldsymbol{x}_\omega \in \prod_{i\in\omega} \mathsf{Dom}(X_i)$.

Equation (5.12) shows that we only need to compute the sum over tuples $\boldsymbol{x}_{\omega_L \setminus \omega_U}$, because after this sum is computed, we can marginalize away variables $\boldsymbol{x}_{\omega_U \setminus \omega_Q}$ in $O(N \log N)$-time. We next show that the inner sum of (5.12) can be computed in poly-logarithmic time.

Fix a tuple $\boldsymbol{x}_{\omega_U}$ such that $\Phi_U(\boldsymbol{x}_{\omega_U}) \neq 0$. A tuple $\boldsymbol{x}_{\omega_L}$ is said to be $\boldsymbol{x}_{\omega_U}$-*adjacent* if $\pi_{U\cap L}\boldsymbol{x}_{\omega_U} = \pi_{U\cap L}\boldsymbol{x}_{\omega_L}$. Then we can rewrite the inner sum of (5.12) as follows:

$$\sum_{\boldsymbol{x}_{\omega_L \setminus \omega_U}} \Phi_U(\boldsymbol{x}_{\omega_U}) \cdot \Phi_L(\boldsymbol{x}_{\omega_L}) \cdot \left(\prod_{S \in \mathcal{E}_\infty} \mathbf{1}_{\sum_{v\in S} \gamma_v^S(x_v) \le 0} \right) = \tag{5.13}$$

$$\Phi_U(\boldsymbol{x}_{\omega_U}) \cdot \sum_{\boldsymbol{x}_{\omega_L \setminus \omega_U}} \Phi_L(\boldsymbol{x}_{\omega_L}) \cdot \left(\prod_{S \in \mathcal{E}_\infty} \mathbf{1}_{\gamma^S_{S\cap\omega_U}(\boldsymbol{x}_{S\cap\omega_U}) \le -\gamma^S_{S\setminus\omega_U}(\boldsymbol{x}_{S\setminus\omega_U})} \right). \tag{5.14}$$

where the inner sum ranges over only tuples $\boldsymbol{x}_{\omega_L}$ which are $\boldsymbol{x}_{\omega_U}$-adjacent; non-adjacent tuples contribute 0.

Now, for each $\boldsymbol{x}_{\omega_U}$ define two k-dimensional points:

$$\boldsymbol{q}(\boldsymbol{x}_{\omega_U}) = (q_S(\boldsymbol{x}_{\omega_U}))_{S\in\mathcal{E}_\infty} \quad \text{where} \quad q_S(\boldsymbol{x}_{\omega_U}) := \gamma^S_{S\cap\omega_U}(\boldsymbol{x}_{S\cap\omega_U}), \tag{5.15}$$

$$\boldsymbol{p}(\boldsymbol{x}_{\omega_L}) = (p_S(\boldsymbol{x}_{\omega_L}))_{S\in\mathcal{E}_\infty} \quad \text{where} \quad p_S(\boldsymbol{x}_{\omega_L}) := -\gamma^S_{S\setminus\omega_U}(\boldsymbol{x}_{S\setminus\omega_U}). \tag{5.16}$$

We write $\boldsymbol{q}(\boldsymbol{x}_{\omega_U}) \preceq \boldsymbol{p}(\boldsymbol{x}_{\omega_L})$ to say that $\boldsymbol{q}(\boldsymbol{x}_{\omega_U})$ is dominated by $\boldsymbol{p}(\boldsymbol{x}_{\omega_L})$ *coordinate-wise*: $q_S(\boldsymbol{x}_{\omega_U}) \le p_S(\boldsymbol{x}_{\omega_L})\ \forall\ S \in \mathcal{E}_\infty$. Assign to each point $\boldsymbol{p}(\boldsymbol{x}_{\omega_L})$ a "weight" of $\Phi_L(\boldsymbol{x}_{\omega_L})$. Now, taking (5.14),

$$\sum_{\boldsymbol{x}_{\omega_L \setminus \omega_U}} \Phi_L(\boldsymbol{x}_{\omega_L}) \cdot \left(\prod_{S \in \mathcal{E}_\infty} \mathbf{1}_{\gamma^S_{S\cap\omega_U}(\boldsymbol{x}_{S\cap\omega_U}) \le -\gamma^S_{S\setminus\omega_U}(\boldsymbol{x}_{S\setminus\omega_U})} \right)$$

$$= \sum_{\boldsymbol{x}_{\omega_L \setminus \omega_U}} \left(\prod_{S \in \mathcal{E}_\infty} \mathbf{1}_{q_S(\boldsymbol{x}_{\omega_U}) \preceq p_S(\boldsymbol{x}_{\omega_L})} \right) \cdot \Phi_L(\boldsymbol{x}_{\omega_L}) \tag{5.17}$$

$$= \sum_{\boldsymbol{x}_{\omega_L \setminus \omega_U}} \mathbf{1}_{\boldsymbol{q}(\boldsymbol{x}_{\omega_U}) \preceq \boldsymbol{p}(\boldsymbol{x}_{\omega_L})} \cdot \Phi_L(\boldsymbol{x}_{\omega_L}). \tag{5.18}$$

The expression thus computes, for a given "query point" $\boldsymbol{q}(\boldsymbol{x}_{\omega_U})$, the weighted sum over all points $\boldsymbol{p}(\boldsymbol{x}_{\omega_L})$ that dominate the query point. This is precisely the *dominance range*

counting problem, which—modulo a $O(N \log^k N)$-preprocessing step—can be solved in time $O(\log^k N)$ [34]. □

5.4.3 Complexity of Evaluating Queries with Additive Inequalities

Equipped with the basic case for two relations, we can now consider the general setting for queries of the the form (5.6).

Theorem 5.18 (Previously presented in [5])**.** *Consider any aggregate query Q of the form* (5.6) *with free variables $\omega_Q \subseteq \mathcal{V}$ and query hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E} = \mathcal{E}_R \cup \mathcal{E}_\infty)$ that is computed over a multi-relational database I, where all relations in I have size at most N. Let d denote the maximum number of additive inequalities covered by a pair of adjacent bags in an optimal relaxed tree decomposition[2], and $k = \max(d-1, 1)$.*

Then the query Q can be answered in time

$$O\left(|\mathcal{V}|^2 \cdot |\mathcal{E}| \cdot s \cdot (N^{\mathsf{r\text{-}fhtw}_{\omega_Q}(\mathcal{H})} + |Q|) \cdot \log^k N\right),$$

where s denotes the number of products in the aggregate expression α.

Proof. We first consider the case of no free variables (i.e. $\omega_Q = \emptyset$), because this case captures the key idea. Fix an optimal relaxed tree decomposition (T, χ). We first compute, for each bag $b \in V(T)$ of the tree decomposition, a relation $B_b(\chi(b))$ such that

$$\begin{aligned} Q(\texttt{SUM}(\alpha)) &= R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}), \phi_1(\omega_{\phi_1}), \ldots, \phi_t(\omega_{\phi_t}) && (5.19)\\ &= (B_b(\chi(b)))_{b \in V(T)}, \phi_1(\omega_{\phi_1}), \ldots, \phi_t(\omega_{\phi_t}). && (5.20) \end{aligned}$$

Recall from Definition 5.5 that every $K \in \mathcal{E}_R$ is covered by at least one bag $\chi(b)$ for $b \in V(T)$. Fix an arbitrary coverage assignment $\beta : \mathcal{E}_s \to V(T)$, where relation R_K is covered by the bag $\chi(\beta(K))$. Then, we can compute the calibrated relations B_b that defines the bag $\chi(b)$ as follows:

$$B_b(\boldsymbol{x}_{\chi(b)}) = \left(R_K(\omega_{R_K})\right)_{K \in \alpha^{-1}(b)}, \left(\pi_{K \cap \chi(b)} R_K(\omega_{R_K})\right)_{K \in \mathcal{E}_R, K \cap \chi(b) \neq \emptyset} \tag{5.21}$$

It is straightforward to verify that (5.20) holds. Using any worst-case optimal join algorithm [97, 96, 133] we can compute (5.21) in time

$$\tilde{O}(N^{\rho^*_{\mathcal{H}}(\chi(b))}). \tag{5.22}$$

Over all $b \in V(T)$, our runtime is bounded by $O(N^w)$, where

$$w = \max_{b \in V(T)} \rho^*_{\mathcal{H}}(\chi(b)). \tag{5.23}$$

[2]Note that d can be a lot smaller than $|\mathcal{E}_\infty|$.

In addition, the size of relation B_b is bounded by N^w. Next we compute (5.20) in time $\tilde{O}(N^w)$. To simplify the notation, we first assume that α is a product of functions $(f_k)_{k\in[p]}$ (i.e., $s = 1$), and generalize the to sums of products result at the end. Under this assumption, we can rewrite (5.20) into an equivalent functional aggregate query (FAQ) expression over semiring $(\mathbb{R}, +, \times)$ [10]. We encode each relation B_b as a factor $\Phi_b : \prod_{X_v\in\chi(b)} \mathsf{Dom}(X_v) \to \{0, 1\}$. For a given tuple $\boldsymbol{x}_{\chi(b)}$ of data values over $\chi(b)$, the function Φ_b can be expressed with the Kronecker delta:

$$\Phi_b(\boldsymbol{x}_{\chi(b)}) = \mathbf{1}_{\boldsymbol{x}_{\omega_{\chi(b)}}\in B_b}$$

In other words, $\Phi_b(\boldsymbol{x}_{\chi(b)}) = 1$, iff $\boldsymbol{x}_{\chi(b)} \in B_b$, and 0 otherwise.

Similarly, for each $j \in [t]$, the inequality ϕ_j encodes a function $\varphi_{\phi_j} : \prod_{X_v\in\omega_{\phi_j}} \mathsf{Dom}(X_v) \to \{0, 1\}$, which can also be encoded with the Kronecker delta. For a given tuple $\boldsymbol{x}_{\omega_{\phi_j}}$ over the variables ω_{ϕ_j} of ϕ_j:

$$\varphi_\phi(\boldsymbol{x}_{\omega_\phi}) = \mathbf{1}_\phi.$$

Under this encoding, the aggregate query (5.20) can be expressed as the following functional aggregate query over the sum-product semiring $(\mathbb{R}, +, \times)$:

$$\varphi_Q() = \sum_{\boldsymbol{x}_\mathcal{V}} \left(\prod_{k\in[p]} f_k(\boldsymbol{x}_{\omega_{f_k}})\right) \left(\prod_{t\in V(T)} \Phi_t(\boldsymbol{x}_{\chi(t)})\right) \cdot \left(\prod_{S\in\mathcal{E}_\infty} \mathbf{1}_{\sum_{v\in S}\gamma_v^S(x_v)\le 0}\right) \tag{5.24}$$

Recall that we assume that each function $f_k(\boldsymbol{x}_{\omega_{f_k}})$ is covered by some bag $t \in V(T)$. Therefore, we can absorb each function f_k in the function Φ_t that defines the bag $\chi(t)$ that covers f_k. This can be computed with the following query,

$$\Phi_t(\boldsymbol{x}_{\chi(t)}) = f_k(\boldsymbol{x}_{\omega_{f_k}}) \cdot \Phi_t(\boldsymbol{x}_{\chi(t)}). \tag{5.25}$$

The absorption of all functions $(f_{jk})_{k\in[p_j]}$ can be done in linear time with respect to the size of the functions $(\Phi_t)_{t\in V(T)}$. Thus, we can simplify (5.24) as follows:

$$\varphi_Q() = \sum_{\boldsymbol{x}_\mathcal{V}} \left(\prod_{t\in V(T)} \Phi_t(\boldsymbol{x}_{\chi(t)})\right) \cdot \left(\prod_{S\in\mathcal{E}_\infty} \mathbf{1}_{\sum_{v\in S}\gamma_v^S(x_v)\le 0}\right). \tag{5.26}$$

We make use of the fact that (T, χ) is a relaxed TD. Fix an arbitrary root of the tree decomposition (T, χ); following the algorithms for factorized aggregate computation, we compute (5.26) by eliminating variables from the leaves of (T, χ) up to the root. Without loss of generality, we assume that the tree decomposition is non-redundant, i.e., no bag is a subset of another in the tree decomposition (otherwise the contained bag factor can be "absorbed" into the containee bag factor). Let t_1 be any leaf of (T, χ), t_2 be its parent,

where $L = \chi(t_1)$ and $U = \chi(t_2)$. We now rewrite (5.26) as follows:

$$\sum_{\boldsymbol{x}_{\mathcal{V}}} \left(\prod_{t \in V(T)} \Phi_t(\boldsymbol{x}_{\chi(t)}) \right) \cdot \left(\prod_{S \in \mathcal{E}_\infty} \mathbf{1}_{\sum_{v \in S} \gamma_v^S(x_v) \leq 0} \right)$$

$$= \sum_{\boldsymbol{x}_{\mathcal{V} \setminus (L \setminus U)}} \sum_{\boldsymbol{x}_{L \setminus U}} \left(\prod_{t \in V(T)} \Phi_t(\boldsymbol{x}_{\chi(t)}) \right) \cdot \left(\prod_{S \in \mathcal{E}_\infty} \mathbf{1}_{\sum_{v \in S} \gamma_v^S(x_v) \leq 0} \right) \tag{5.27}$$

$$= \sum_{\boldsymbol{x}_{\mathcal{V} \setminus (L \setminus U)}} \left(\prod_{t \in V(T) \setminus \{t_1, t_2\}} \Phi_t(\boldsymbol{x}_{\chi(t)}) \right) \cdot \left(\prod_{\substack{S \in \mathcal{E}_\infty \\ S \cap (L \setminus U) = \emptyset}} \mathbf{1}_{\sum_{v \in S} \gamma_v^S(x_v) \leq 0} \right)$$

$$\cdot \underbrace{\left[\sum_{\boldsymbol{x}_{L \setminus U}} \Phi_{t_1}(\boldsymbol{x}_L) \cdot \Phi_{t_2}(\boldsymbol{x}_U) \cdot \left(\prod_{\substack{S \in \mathcal{E}_\infty \\ S \cap (L \setminus U) \neq \emptyset}} \mathbf{1}_{\sum_{v \in S} \gamma_v^S(x_v) \leq 0} \right) \right]}_{\text{sub-query of the form (5.9)}} . \tag{5.28}$$

The third equality uses the semiring's distributive law. (Note that $S \cap (L \setminus U) \neq \emptyset$ implies that $S \subseteq (L \cup U)$ thanks to Definition 5.5 and the fact that t_2 is the only neighbor of t_1.) Lemma 5.17 implies that we can compute the sub-query in time $O(N \cdot log^k N)$. The above step eliminates all variables in $L \setminus U$. Repeatedly applying the above step yields the desired output $Q()$. When the query has free variables, the algorithm proceeds similarly to the case of an FAQ query with free variables [11, 10]. Overall this query can than be computed in time:

$$O\left(|\mathcal{V}|^2 \cdot |\mathcal{E}| \cdot (N^{\mathsf{r\text{-}fhtw}_{\omega_Q}(\mathcal{H})} + |Q|) \cdot \log^k N \right),$$

which follows from the runtime of the factorized query evaluation algorithms [10].

If α encodes a sum of s products of functions, then we compute the above query s times for each product of functions, and then sum up the result. This completes the proof. □

Recall the lower bound for r-fhtw from Proposition 5.14. The lower bound shows that, while computing aggregate queries of the form (5.6) over relaxed tree decompositions can improve the runtime by a polynomial factor, it cannot improve the complexity of evaluating aggregate queries over non-relaxed tree decompositions as shown in Theorem 5.15 by more than a polynomial factor.

We next give a simple upper bound, which shows that, effectively r-fhtw is the best we can hope for, for arbitrary queries of the form (5.6).

Proposition 5.19. *For any positive integer m, there exists a query of the form* (5.6) *with hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E} = \mathcal{E}_R \cup \mathcal{E}_\infty)$ for which $\omega_Q = \emptyset$, $\mathsf{r\text{-}fhtw}(Q) \geq m$ and it cannot be answered in time $o(N^{\mathsf{r\text{-}fhtw}(\mathcal{H})})$, unless there exists an algorithm that can solve the k-sum problem in time $O(N^{\lceil k/2 \rceil - \gamma})$, for any $\gamma > 0$.*

Proof. It is widely assumed that $O(N^{\lceil k/2 \rceil})$ is the best runtime for k-sum [104, 81], which is the following problem: given k number sets $R_1, \ldots, R_k$ of maximum size N, determine

whether there is a tuple $\boldsymbol{t} \in R_1 \times \cdots \times R_k$ such that $\sum_{i\in[k]} t_i = 0$. We can reduce k-sum to our problem. Consider the following boolean conjunctive query:

$$Q() \leftarrow R_1(x_i), \ldots, R_k(x_k), \sum_{i\in[k]} x_i \leq 0, \sum_{i\in[k]} x_i \geq 0. \tag{5.29}$$

The answer to Q is true iff there is a tuple $(x_1, \ldots, x_k) \in R_1 \times \cdots \times R_k$ such that $\sum_{i\in[k]} x_i = 0$. The reduction shows that our query (5.29) is k-sum-hard. For this query, $\mathsf{r\text{-}fhtw}(\mathcal{H}) = \lceil k/2 \rceil$. □

5.5 Computing Batches of Aggregate Queries

In the previous sections, we consider the case of evaluating a single aggregate query. Our structure aware approach to learning over databases, however, requires the computation of many aggregate queries. We next show that, for a large class of feature extraction queries, we can also compute a batch of aggregate queries asymptotically faster than the time it takes to materialize the feature extraction query. This result is central to our theoretical analysis for learning models over multi-relational databases.

Let I be a database with relations $R_1, \ldots, R_m$, where all relations have size at most N. We assume that the feature extraction query φ is defined by the join of relations in I. We further consider a batch of $\ell \geq 1$ aggregate queries $(Q_i)_{i\in[\ell]}$ that are computed over the feature extraction query. Each query Q_i has group-by variables ω_{Q_i}. We assume that all queries have at most $k = \max_i |\omega_{Q_i}|$ group-by variables, and that each query defines a single aggregate that is a product of UDAFs (i.e., $s = 1$). Since the feature extraction query φ and all queries $(Q_i)_{i\in[\ell]}$ share the same query body, they all have the same query hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$.

The following proposition shows that there is a large class of feature extraction queries for which the entire aggregate query batch can be computed asymptotically faster than the time it takes to materialize the feature extraction query.

Proposition 5.20. *We can compute the entire batch of aggregate queries in time:*

$$O\left(|\mathcal{V}|^2 \cdot |\mathcal{E}| \cdot \sum_{i\in[\ell]} (N^{\mathsf{fhtw}_{\omega_{Q_i}}(\mathcal{H})} + N^k) \cdot \log N\right)$$

For any feature extraction query with $\rho^*_{\mathcal{H}}(\mathcal{V}) > \mathsf{fhtw}(\mathcal{H}) + k - 1$, the following holds:

$$\lim_{N\to\infty} \frac{N^{\rho^*(\mathcal{V})}}{\sum_{i\in[\ell]} (N^{\mathsf{fhtw}_{\omega_{Q_i}}(\mathcal{H})} + N^k) \cdot \log N} = \infty. \tag{5.30}$$

Proof. Each query Q_i can have at most k free variables, and the active domain for each variable has size at most N. Thus, it follows that $|Q_i| \leq N^k$. The runtime bound for the entire query batch follows directly from Theorem 5.15.

Fix a feature extraction query φ with $\rho^*_{\mathcal{H}}(\mathcal{V}) > \mathsf{fhtw}(\mathcal{H}) + k - 1 \geq k$. Equation (5.30) follows from the fact that there exists a database instance I for which $|\varphi| = \Theta(N^{\rho^*})$, which is guaranteed by the lower bound of the AGM bound [19, 60]. □

5.6 Discussion: Beyond Fractional Hypertree Width

There exist width measures for a hypergraph $\mathcal{H}$ that can be lower than the fractional hypertree width fhtw of $\mathcal{H}$. Two such width measures are the submodular width (subw) and the #-submodular width ($\#\mathsf{subw}$). Recent developments in database query evaluation have shown that there also exist two algorithms PANDA and #PANDA that can compute conjunctive queries in time given by subw and respectively aggregate queries in time given by $\#\mathsf{subw}$. As a result, these algorithms can have asymptotically lower runtime in data complexity than the factorized query evaluation algorithms considered in this chapter.

We next give the definitions of the submodular width and the #-submodular width. We then present runtime bounds for PANDA and #PANDA, and a short high-level explanation why they achieve a better runtime than the factorized query evaluation algorithms considered above. Then, we discuss why we do not consider these algorithms in this thesis.

Submodular width

To define the submodular width and the #-submodular width parameters, we use the polymatroid characterization from [75, 5].

Definition 5.21 (Adapted from [5])**.** Given a collection $\mathcal{E}$ of subsets of $\mathcal{V}$, a set function $h : 2^{\mathcal{V}} \to \mathbb{R}_+$ is said to be a *$\mathcal{E}$-polymatroid* if it satisfies the following: (i) $h(\emptyset) = 0$, (ii) $h(X) \leq h(Y)$ whenever $X \subseteq Y$, and (iii) $h(X \cup Y) + h(X \cap Y) \leq h(X) + h(Y)$ for every pair $X, Y \subseteq \mathcal{V}$ such that $X \cap Y \subseteq S$ for some $S \in \mathcal{E}$.

For $\mathcal{V} = [n]$, we use $\Gamma_{n|\mathcal{E}}$ to denote the set of all $\mathcal{E}$-polymatroids on $\mathcal{V}$. □

Given a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E} = \mathcal{E}_R \cup \mathcal{E}_\infty)$, we define the set of *edge dominated* set functions:

$$\mathsf{ED} := \{h \mid h : 2^{\mathcal{V}} \to \mathbb{R}_+, h(F) \leq 1, \forall F \in \mathcal{E}_R\}. \tag{5.31}$$

We can now define the submodular width of a given hypergraph $\mathcal{H}$.

Definition 5.22 (Adapted from [75, 9])**.** Given a multi-hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E} = \mathcal{E}_R \cup \mathcal{E}_\infty)$,

the *submodular width* is defined by

$$\mathsf{subw}(\mathcal{H}) := \max_{h \in \mathsf{ED} \cap \Gamma_{n|2}\mathcal{V}} \min_{(T,\chi)\in \mathsf{TD}} \max_{t \in V(T)} h(\chi(t)).$$

For $F \subseteq \mathcal{V}$, the F-connex *submodular width* is defined by

$$\mathsf{subw}_F(\mathcal{H}) := \max_{h \in \mathsf{ED} \cap \Gamma_{n|2}\mathcal{V}} \min_{(T,\chi)\in \mathsf{TD}_F} \max_{t \in V(T)} h(\chi(t)).$$

□

The following definition is a straightforward adaptation of subw, where we replace $\Gamma_{n|\mathcal{V}}$ by the set of all $\mathcal{E}_R$-polymatroids of $\mathcal{V}$.

Definition 5.23 (Adapted from [5])**.** Given a multi-hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E} = \mathcal{E}_R \cup \mathcal{E}_\infty)$ and $F \subseteq \mathcal{V}$, the F-connex *#-submodular width* is defined by

$$\#\mathsf{subw}(\mathcal{H}) := \max_{h \in \mathsf{ED} \cap \Gamma_{n|\mathcal{E}_R}} \min_{(T,\chi)\in \mathsf{TD}} \max_{t \in V(T)} h(\chi(t)).$$

For $F \subseteq \mathcal{V}$, the F-connex *#-submodular width* is defined by:

$$\#\mathsf{subw}_F(\mathcal{H}) := \max_{h \in \mathsf{ED} \cap \Gamma_{n|\mathcal{E}_R}} \min_{(T,\chi)\in \mathsf{TD}_F} \max_{t \in V(T)} h(\chi(t)).$$

□

It is known that $\mathsf{subw}_F(\mathcal{H}) \leq \mathsf{fhtw}_F(\mathcal{H})$, and there are classes of hypergraphs with bounded subw and unbounded fhtw [85].

Analogous to $\mathsf{r\text{-}fhtw}$, we can also define relaxed variants of subw and $\#\mathsf{subw}$, which are defined over the set of all relaxed tree decompositions [5].

Remark 1. In Definition 5.10, we defined the fractional hypertree width of an hypergraph $\mathcal{H}$ based on the the fractional edge cover number:

$$\mathsf{fhtw}(\mathcal{H}) = \min_{(T,\chi)\in \mathsf{TD}(\mathcal{H})} \max_{t \in V(T)} \rho^*_{\mathcal{H}}(\chi(t)).$$

For a given given tree decomposition (T, χ), it has been shown, using linear programming duality, that [75]:

$$\max_{t \in V(T)} \rho^*_{\mathcal{E}}(\chi(t)) = \max_{t \in V(T)} \max_{h \in \mathsf{ED} \cap \Gamma_n} h(\chi(t)).$$

Therefore, an equivalent definition for $\mathsf{fhtw}(\mathcal{H})$ is given by:

$$\mathsf{fhtw}(\mathcal{H}) = \min_{(T,\chi)\in \mathsf{TD}} \max_{h \in \mathsf{ED} \cap \Gamma_{n|\mathcal{V}}} \max_{t \in V(T)} h(\chi(t)). \tag{5.32}$$

This alternative characterization of fhtw exposes the minimax/maximin duality between fhtw and subw. □

Runtime of PANDA and #PANDA

The PANDA algorithm can be used to evaluate conjunctive queries, i.e., queries of the form (4.1) without aggregates. In contrast to the factorized query evaluation algorithm discussed in this chapter, PANDA does not compute the query result over a single tree decomposition. Instead, it uses information theoretic inequalities to decompose the query evaluation problem into smaller subproblems, where each subproblem is the computed over its respective optimal tree decompositions. The following theorem gives the runtime of PANDA.

Theorem 5.24 (Adapted from [75])**.** *Given a conjunctive query with hypergraph* $\mathcal{H} = (\mathcal{V}, \mathcal{E} = \mathcal{E}_R \cup \mathcal{E}_\infty)$ *and free variables* $F \subseteq \mathcal{V}$ *over a database of size* N*, the* PANDA *algorithm can answer* φ *in time:*

$$O\left(|\mathcal{V}|^2 \cdot 2^{2^{|\mathcal{V}|}} \cdot (N^{\mathsf{subw}_F(\mathcal{H})} \cdot \mathsf{poly}(\log N) + |\varphi| \cdot \log N)\right).$$

□

The PANDA algorithm does not support queries with aggregates, because the subproblems derived by the algorithm can overlap [5]. Thus, when PANDA decomposes the problem into subproblem that overlap, then it would have to perform inclusion-exclusion over the results of the different subproblems, which would explode the runtime.

The #PANDA algorithm is an adaptation of PANDA which ensures that the subproblems are disjoint, and therefore it can evaluate queries with aggregates [5]. The runtime of #PANDA is given by the following theorem.

Theorem 5.25 (Adapted from [5])**.** *Given an aggregate query* φ *with hypergraph* $\mathcal{H} = (\mathcal{V}, \mathcal{E} = \mathcal{E}_R \cup \mathcal{E}_\infty)$ *over a database of size* N*, the* #PANDA *algorithm can answer* φ *in time:*

$$O\left(|\mathcal{V}|^2 \cdot 2^{2^{|\mathcal{V}|}} \cdot (N^{\#\mathsf{subw}_F(\mathcal{H})} \cdot \mathsf{poly}(\log N) + |\varphi| \cdot \log N)\right).$$

□

For queries with additive inequalities, PANDA and #PANDA can be modified to compute the queries over relaxed tree decompositions, and thus in time given by the relaxed submodular width and respectively the relaxed #-submodular width [5].

Discussion

In principle, we could use PANDA and #PANDA to compute queries of the form (4.1). In this thesis, however, we focus on the factorized query evaluation algorithms, because we believe that they perform better in practice. Our rationale is twofold.

First, the runtime for the factorized query evaluation algorithms does not have the large $\mathsf{poly}(\log N)$ and $2^{2^{|\mathcal{V}|}}$ factors, which in practice might easily outweigh the benefit of the lower data complexity.

In addition, using multiple tree decompositions for different subproblems precludes the algorithm from sharing computation between subproblems. In practice, however, we found that the sharing of computation can lead to significant performance improvements. For this reason, the LMFAO engine presented in Chapter 11, goes even further and uses the same tree decomposition for multiple queries, which leads to significant performance improvements over state-of-the-art data managements systems.

Chapter 6

Optimization Problems with Square Loss Function

We consider the problem of learning a supervised machine learning model[1] $\mathcal{M}_{\boldsymbol{\theta}}(\boldsymbol{x})$ with parameters $\boldsymbol{\theta}$ over a training dataset D. Each tuple in $(\boldsymbol{x}, y) \in D$ provides a feature vector $\boldsymbol{x}$ and label y. Once learned, the model $\mathcal{M}_{\boldsymbol{\theta}}(\boldsymbol{x})$ is used to predict the label y for an unlabeled data point $\boldsymbol{x}$. The parameters are learned by minimizing the objective function over D:

$$J(\boldsymbol{\theta}) = \sum_{(\boldsymbol{x},y)\in D} \mathcal{L}\left(\mathcal{M}_{\boldsymbol{\theta}}(\boldsymbol{x}), y\right) + \lambda\Omega(\boldsymbol{\beta}), \tag{6.1}$$

where $\mathcal{L}(a, b)$ is a loss function, Ω is a regularizer, e.g., ℓ_1 or ℓ_2 norm, and the constant λ controls the influence of regularization.

In this chapter, we consider the class of supervised machine learning models, where the objective function $J(\boldsymbol{\theta})$ is defined by the square loss function:

$$\mathcal{L}(a, b) = (a - b)^2. \tag{6.2}$$

This class includes (ridge and lasso) linear regression, polynomial regression, and factorization machines. The optimization problem is then defined as follows.

We assume that the training dataset D is the result of a feature extraction query Q computed over database I with relations $R_1, \ldots, R_m$. The query Q has variables $X_1, \ldots, X_n$ and label Y. The goal is to learn the parameters $\boldsymbol{\theta}$ of parameterized model $\mathcal{M}_{\boldsymbol{\theta}}(\boldsymbol{x})$ over the training dataset D, by minimizing the least-squares objective function:

$$J(\boldsymbol{\theta}) = \frac{1}{2|D|} \sum_{(\boldsymbol{x},y)\in D} \left(\mathcal{M}_{\boldsymbol{\theta}}(\boldsymbol{x}) - y\right)^2 + \lambda\Omega(\boldsymbol{\theta}), \tag{6.3}$$

where Ω is a regularization term, and λ is a constant. For a tuple $(\mathbf{x}, y) \in D$, the square

[1] For this thesis, we focus on the learning task,and thus do not discuss model testing and deployment.

loss function gives the error of the model $\mathcal{M}_{\boldsymbol{\theta}}(\boldsymbol{x})$ with respect to the true label y.

The regularization term $\Omega(\boldsymbol{\theta})$ penalizes the complexity of the model in order to avoid overfitting. A model is overfitting when it is fit too closely to the training datasets, and performs poorly on a test dataset. Two common regularization functions are (1) the squared ℓ_2-norm $\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|\boldsymbol{\theta}\|_2^2$; and (2) the ℓ_1-norm $\Omega(\boldsymbol{\theta}) = \|\boldsymbol{\theta}\|_1$.

There exist many optimization algorithms that can be used to minimize $J(\boldsymbol{\theta})$. In this chapter, we consider batch gradient descent optimization. Section 6.7 overviews alternative optimization algorithms for this problem, including stochastic gradient descent, coordinate descent, and Quasi-Newton methods.

We show that the data intensive computation of optimization problems with objective (6.3) can be expressed as batches of aggregate queries of the form (4.1) that compute a single aggregate over the join of the relations in I:

$$Q(\omega_F, \texttt{SUM(}\alpha\texttt{)}) \leftarrow R_1(\omega_1), \ldots, R_m(\omega_m),$$

where $\omega_F \subseteq \{X_1, \ldots X_n, Y\}$ and α is a sum of products of user-defined aggregate functions. We show the rewriting of the learning problem into aggregate queries for learning linear regression models, polynomial regression models, and factorization machines.

This rewriting has several advantages. (1) We can factorize the computation of the aggregate queries, and compute the end-to-end pipeline for learning over databases exponentially faster than current state-of-the-art machine learning systems. (2) The aggregate queries naturally accommodate for categorical variables. In contrast, most machine learning systems require such variables to be one-hot encoded in a preprocessing step. (3) We can efficiently support model selection, which is a powerful but time consuming process that aims to find the model defined by a subset of features that best predicts the label.

The results presented in this chapter have previously been published in [7, 8, 119, 101, 100], and the background information is mainly adapted from [92].

Chapter Outline

The outline for this chapter is as follows:

- Sections 6.1 and 6.2 give a short introduction to one-hot encoding of categorical features and respectively batch gradient descent optimization.
- Section 6.3 presents our solution and its complexity analysis for learning linear regression models.
- Section 6.4 extends the solution and complexity analysis for linear regression models to polynomial regression models of arbitrary degree.
- Section 6.5 shows how the solution for polynomial regression models can be extended to learn factorization machines.

- Section 6.6 exemplifies how our solution can be used for models selection.
- Section 6.7 overviews alternative optimization algorithms, including stochastic gradient descent, coordinate descent, and Quasi-Newton optimization algorithms (e.g., L-BFGS). We compare these algorithms to batch gradient descent, and explain to what extend their data intensive computation can be expressed as aggregate queries.

6.1 Primer: One-Hot Encoding of Categorical Features

Many machine learning models classify the input variables $\boldsymbol{X}$ as continuous and categorical variables. Continuous variables are quantitative variables, such as age, salary, or price. Categorical variables are qualitative variables, such as colors, cities, and countries. The active domain of a categorical variable is a set of possible values or categories, i.e. germany, england, and usa are possible categories of the categorical feature country.

Example 6.1. Recall the feature extraction query from Example 4.2:

$$Q(\mathsf{sku}, \mathsf{store}, \mathsf{color}, \mathsf{quarter}, \mathsf{city}, \mathsf{country}, \mathsf{units\ sold}, \mathsf{price}, \mathsf{size})$$
$$\leftarrow \mathsf{Sales}(\mathsf{sku}, \mathsf{store}, \mathsf{day}, \mathsf{units\ sold}), \mathsf{Items}(\mathsf{sku}, \mathsf{color}, \mathsf{price}), \mathsf{Quarter}(\mathsf{day}, \mathsf{quarter}),$$
$$\mathsf{Stores}(\mathsf{store}, \mathsf{city}, \mathsf{size}), \mathsf{Country}(\mathsf{city}, \mathsf{country}).$$

In this query, all variables except price, size, and the label units sold are categorical variables. □

While continuous variables allow for aggregation over their domains, categorical variables cannot be aggregated together. Most machine learning models require a preprocessing of such variables before learning the model. The state-of-the-art approach is to one-hot encode the active domain of each categorical variable [92]: each value in the active domain of a variable is encoded by an indicator vector whose dimension is the size of the domain.

Example 6.2. The colors in the domain {red, green, blue} can be represented by indicator vectors $[1, 0, 0]$ for red, $[0, 1, 0]$ for green, and $[0, 0, 1]$ for blue. □

The one-hot encoding amounts to a relational representation of the training dataset with one new variable per distinct category of each categorical variable and with wide tuples whose values are mostly 0. This entails huge redundancy due to the presence of the many 0 values. The one-hot encoding also blurs the usual distinction between schema and data, since the schema can become as large as the input database.

We use the following convention to uniformly encode continuous and categorical variables. Let D denote a training dataset with variables $\boldsymbol{X} = (X_1, \ldots, X_n)$ and label Y, which is defined by a feature extraction query Q. For each tuple $(\boldsymbol{x}, y) \in D$, the vector $\boldsymbol{x} = (\boldsymbol{x}_i)_{i \in [n]}$ over variables $\boldsymbol{X}$ encodes a feature vector for the model, and y is a value for

optimize (parameters $\boldsymbol{\theta}$, objective $J(\boldsymbol{\theta})$)

while (`not converged`) **do** {
  $\boldsymbol{d} \leftarrow \boldsymbol{\nabla} J(\boldsymbol{\theta})$;
  $s \leftarrow$ `next step size`;
  `/* Backtracking Line Search */`
  while $(J(\boldsymbol{\theta} - s\boldsymbol{d}) \geq J(\boldsymbol{\theta}) + \frac{s}{2} \|\boldsymbol{d}\|_2^2)$ **do**
    $s \leftarrow s/2$;
  $\boldsymbol{\theta} = \boldsymbol{\theta} - s\boldsymbol{d}$;
}

Algorithm 6.1: Template for the batch gradient descent optimization algorithm of an objective function $J(\boldsymbol{\theta})$.

the label Y. For each variable $(X_i)_{i\in[n]}$, the component $\boldsymbol{x}_i$ of the feature vector $\boldsymbol{x}$ is also a vector. If X_i is a categorical variable, then $\boldsymbol{x}_i$ has indicator value for each $v \in \mathsf{Dom}(X_j)$: $\boldsymbol{x}_j = (\mathbf{1}_{X_j=v})_{v\in\mathsf{Dom}(X_j)}$. For continuous variable X_i, the vector $\boldsymbol{x}_i = [v]$ simply contains one scalar which is a value $v \in \mathsf{Dom}(X_i)$.

Example 6.3. Consider the feature extraction query from Example 6.1. For each tuple $(\boldsymbol{x}, y)$ in the query result, the vector $\boldsymbol{x}$ encodes an eight-dimensional vector, where each component is another vector that corresponds to one feature in $\boldsymbol{X} = \{$sku,store,color,quarter,city, country,price,size$\}$.

For continuous variable price, the corresponding component $\boldsymbol{x}_{\mathsf{price}}$ in $\boldsymbol{x}$ is a one-dimensional vector, which returns the value of price: $\boldsymbol{x}_{\mathsf{price}} = (\mathsf{price})$.

The component vectors for categorical variables, however, contain one value for each category in the domain of that variable. For instance, assuming the domain of color is {red, green, blue}, then the component $\boldsymbol{x}_{\mathsf{color}}$ is a three dimensional vector with one indicator value per category:

$$\boldsymbol{x}_{\mathsf{color}} = (\mathbf{1}_{\mathsf{color}=\mathrm{red}}, \mathbf{1}_{\mathsf{color}=\mathrm{green}}, \mathbf{1}_{\mathsf{color}=\mathrm{blue}}).$$

Thus, if color is red, then the component is $\boldsymbol{x}_{\mathsf{color}}$ in $\boldsymbol{x}$ is the vector $\boldsymbol{x}_{\mathsf{color}} = (1, 0, 0)$. □

6.2 Primer: Batch Gradient Descent Optimization

We use the batch gradient descent (BGD) optimization algorithm to optimize parameters $\boldsymbol{\theta}$ of a given objective function $J(\boldsymbol{\theta})$. BGD employs the first-order gradient information for the optimization: It repeatedly updates the parameters $\boldsymbol{\theta}$ by step size s in the direction of the gradient vector $\boldsymbol{\nabla} J(\boldsymbol{\theta})$. The gradient vector is the vector of partial derivatives, i.e., the k'th component of $\boldsymbol{\nabla} J(\boldsymbol{\theta})$ is the partial derivative $\frac{\partial}{\partial \theta_k} J(\boldsymbol{\theta})$ with respect to the parameter θ_k. To guarantee convergence, it uses a backtracking line search to ensure that step size s is sufficiently small to decrease the loss function. Algorithm 6.1 presents the pseudocode

for BGD with the Armijo line search condition [17].

When combined with backtracking line search, BGD is guaranteed to converge to a minimum with linear asymptotic convergence rate. Without loss of generality, we assume that the number of iterations of BGD is bounded. (This bound is typically dimension-free, dependent on the Lipschitz constant of J.)

We use the Barzilai-Borwein step size adjustment [24] to choose the next step size, but there also exist many other approaches [92, 55, 46]. In the following, we focus on the computation of $J(\boldsymbol{\theta})$ and $\boldsymbol{\nabla} J(\boldsymbol{\theta})$, which represents the data-intensive computation and main bottleneck of the algorithm.

In case the regularizer in the objective $J(\boldsymbol{\theta})$ is non-differentiable, such as the ℓ_1-norm, then the gradient vector $\boldsymbol{\nabla} J(\boldsymbol{\theta})$ is not defined. In many cases, the objective, however, admits subgradient vectors, which generalize the standard notion of gradients. Then, we can still optimize the object $J(\boldsymbol{\theta})$ with Algorithm 6.1, where the parameters are updated in the direction of the subgradient of $J(\boldsymbol{\theta})$ instead.

6.3 Linear Regression

Let D denote a training dataset with variables $\boldsymbol{X} = (X_1, \ldots, X_n)$ and label Y, which is defined by a feature extraction query Q over a database with relations $R_1, \ldots, R_m$.

Linear regression models require that categorical variables are one-hot encoded. We therefore encode the features of the model using the uniform encoding for continuous and categorical variables presented in Section 6.1. The model then assigns a parameter θ to each feature in the model. To match our encoding of the feature vector $\boldsymbol{x}$, we encode the parameters as a vector $\boldsymbol{\theta} = (\boldsymbol{\theta}_i)_{i \in [n]}$, where each component $\boldsymbol{\theta}_i$ is a vector of scalar parameters of the same dimension as $\boldsymbol{x}_i$. For a categorical variable X_i, the vector $\boldsymbol{\theta}_i = (\theta_v)_{v \in \mathsf{Dom}(X_i)}$ thus stores one parameter for each value in the domain of X_i. For a continuous variable X_i, the component $\boldsymbol{\theta}_i = [\theta_{X_i}]$ has a single parameter for X_i.

Given a feature vector $\boldsymbol{x} = (\boldsymbol{x}_i)_{i \in [n]}$ over variables $\boldsymbol{X}$, a linear regression model with parameters $\boldsymbol{\theta}$ is defined as follows:

$$\mathsf{LR}(\boldsymbol{x}) = \sum_{j \in [n]} \boldsymbol{\theta}_j^\top \boldsymbol{x}_j = \langle \boldsymbol{\theta}, \boldsymbol{x} \rangle \,, \tag{6.4}$$

where $\langle \boldsymbol{\theta}, \boldsymbol{x} \rangle$ encodes the Frobenius inner product.

We assume, without loss of generality, that $\boldsymbol{x}_1$ always takes value 1, so that $\boldsymbol{\theta}_1$ is the intercept of the model.

Example 6.4. We consider a common retail analytics scenario, where the aim is to predict future sales for a given item at one of the stores. The training dataset is defined by a feature extraction query over the following three relations: (1) $\mathsf{Sales}(\mathsf{item}, \mathsf{store}, \mathsf{units\ sold})$, which stores for each item and store the number of units sold; (2) $\mathsf{Items}(\mathsf{item}, \mathsf{price}, \mathsf{color})$, which

keeps the price and color of each item; and (3) $\mathsf{Stores}(\mathsf{store}, \mathsf{city}, \mathsf{size})$, which provides the size of the store as well as the city that the store is located in.

$$Q(\mathsf{price}, \mathsf{size}, \mathsf{city}, \mathsf{color}, \mathsf{units\ sold})$$
$$\leftarrow \mathsf{Sales}(\mathsf{item}, \mathsf{store}, \mathsf{units\ sold}), \mathsf{Items}(\mathsf{item}, \mathsf{price}, \mathsf{color}), \mathsf{Stores}(\mathsf{store}, \mathsf{city}, \mathsf{size}).$$

We assume that the feature extraction query projects away item and store, and that there are three cities (london, oxford, bristol) and two colors (blue, red) in the dataset. We encode the values for city and color as indicator values. For instance, we let $\mathsf{oxford} = \mathbf{1}_{\mathsf{city}=\mathsf{oxford}}$, which is 1 if $\mathsf{city} = \mathsf{oxford}$, and 0 otherwise. Similarly, we let $\mathsf{blue} = \mathbf{1}_{\mathsf{color}=\mathsf{blue}}$ which is 1 if $\mathsf{color} = \mathsf{blue}$, and 0 otherwise. The label for the model is given by units sold.

The linear regression model would then be a model with eight parameters:

$$\mathsf{LR}(\mathsf{price}, \mathsf{size}, \mathsf{city}, \mathsf{color}) =$$
$$\theta_{\mathsf{intrcpt}} + \theta_{\mathsf{price}} \cdot \mathsf{price} + \theta_{\mathsf{size}} \cdot \mathsf{size} + \begin{bmatrix}\theta_{\mathsf{oxford}} & \theta_{\mathsf{london}} & \theta_{\mathsf{bristol}}\end{bmatrix} \begin{bmatrix}\mathsf{oxford} \\ \mathsf{london} \\ \mathsf{bristol}\end{bmatrix} + \begin{bmatrix}\theta_{\mathsf{blue}} & \theta_{\mathsf{red}}\end{bmatrix} \begin{bmatrix}\mathsf{blue} \\ \mathsf{red}\end{bmatrix}$$

where $\theta_{\mathsf{intrcpt}}$ encodes the intercept of the model. The features for continuous variables price and size are scalars, whereas the features for categorical variables city and color as encoded as vectors of indicator values. □

Linear regression as an optimization problem

The parameters of a linear regression model are learned by minimizing an objective function $J(\boldsymbol{\theta})$ from Equation (6.3). If the regularization term is give by the ℓ_2 norm, $\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|\boldsymbol{\theta}\|_2^2$, then the model is called a *ridge* regression model. If it is the ℓ_1-norm, $\Omega(\boldsymbol{\theta}) = \|\boldsymbol{\theta}\|_1$, then the model is called a *lasso* regression model. In the following, we consider the case where $J(\boldsymbol{\theta})$ is regularized by the ℓ_2-norm:

$$J(\boldsymbol{\theta}) = \frac{1}{2|D|} \sum_{(\boldsymbol{x},y)\in D} \Big(\sum_{j\in[n]} \boldsymbol{\theta}_j^\top \boldsymbol{x}_j - y \Big)^2 + \frac{\lambda}{2} \|\boldsymbol{\theta}\|_2^2 \tag{6.5}$$

We extend, without loss of generality, the feature vector $\boldsymbol{x} = (\boldsymbol{x}_i)_{i\in[n]}$ with an additional term $\boldsymbol{x}_{n+1} = [y]$, which consists of a single value for the label y of the given feature vector. We also define a corresponding new parameter $\boldsymbol{\theta}_{n+1}$ which always takes value -1. Then, we can simplify for the expression of the objective function:

$$J(\boldsymbol{\theta}) = \frac{1}{2|D|} \sum_{\boldsymbol{x}\in D} \Big(\sum_{j\in[n+1]} \boldsymbol{\theta}_j^\top \boldsymbol{x}_j \Big)^2 + \frac{\lambda}{2} \|\boldsymbol{\theta}\|_2^2 \tag{6.6}$$

To minimize $J(\boldsymbol{\theta})$, we use BGD optimization as shown in Algorithm 6.1. For each

$k \in [n]$, BGD repeatedly updates parameter $\boldsymbol{\theta}_k$ in the direction of:

$$\frac{\partial}{\partial \boldsymbol{\theta}_k} J(\boldsymbol{\theta}) = \frac{1}{|D|} \sum_{\boldsymbol{x} \in D} \left(\sum_{j \in [n+1]} \boldsymbol{\theta}_j^\top \boldsymbol{x}_j \right) \boldsymbol{x}_k^\top + \lambda \boldsymbol{\theta}_k. \tag{6.7}$$

Computing $\nabla J(\boldsymbol{\theta})$ using aggregate queries

We next focus on the data dependent computation of $\frac{\partial}{\partial \boldsymbol{\theta}_k} J(\boldsymbol{\theta})$ from Equation (6.7), and show that this can be expressed as aggregate queries over the input database. There are two ways to express these aggregate queries.

The first approach directly computes the quantity $G_k = \sum_{\boldsymbol{x} \in D} \sum_{j \in [n+1]} (\boldsymbol{\theta}_j^\top \boldsymbol{x}_j) \boldsymbol{x}_k^\top$ for each $k \in [n]$. To express G_k as an aggregate query in the query language presented in Section 4.2, we define for each $j \in [n+1]$ a unary function f_j that encodes the product $\boldsymbol{\theta}_j^\top \boldsymbol{x}_j$. Consider a given tuple $(\boldsymbol{x}, y) \in D$. If X_j is a continuous feature, then the corresponding function $f_j(x_j) = \theta_{X_j} \cdot x_j$ computes the product of the scalar parameter θ_{X_j} with the value x_j. If X_j is a categorical feature, recall that one-hot encoding implies that the vector $\boldsymbol{x}_j$ is a vector of indicator values where the entry for $x_j \in \mathsf{Dom}(X_j)$ is equal to 1 and the rest 0. Thus, the product $\boldsymbol{\theta}_j^\top \boldsymbol{x}_j$ results in the parameter in $\boldsymbol{\theta}_j$ that corresponds to the value x_j. This can be encoded as a unary function $f_j(x_j) = \boldsymbol{\theta}_{X_j}(x_j)$. If $j = n + 1$, then $f_j(y) = -y$.

Example 6.5. For the linear regression model from Example 6.4, the functional encoding for price is the function $f(\mathsf{price}) = \theta_{\mathsf{price}} \cdot \mathsf{price}$. For city, the product $\boldsymbol{\theta}_{\mathsf{city}}^\top \boldsymbol{x}_{\mathsf{city}}$ can be encoded by the function:

$$f_{\mathsf{city}}(\mathsf{city}) = \begin{cases} \theta_{\mathsf{oxford}} & \text{if } \mathsf{city} = \mathsf{oxford} \\ \theta_{\mathsf{london}} & \text{if } \mathsf{city} = \mathsf{london} \\ \theta_{\mathsf{bristol}} & \text{if } \mathsf{city} = \mathsf{bristol} \end{cases}$$

□

Using the functional encoding of the products $\boldsymbol{\theta}_j^\top \boldsymbol{x}_j$, we can now define aggregate queries which compute G_k. The exact specification of the query depends on the type of feature X_k. If X_k is a continuous feature, then G_k computes a scalar which can be represented with the following query:

$$\mathsf{Partial}_k \left(\mathtt{SUM}\big(\textstyle\sum_{j \in [n+1]} f_j(X_j) \cdot X_k \big) \right) \leftarrow R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}). \tag{6.8}$$

If X_k is categorical, then G_k computes a vector with one entry for each $v \in \mathsf{Dom}(X_k)$. An equivalent relational encoding of this vector can be computed with the following query:

$$\mathsf{Partial}_k \left(X_k, \mathtt{SUM}\big(\textstyle\sum_{j \in [n+1]} f_j(X_j) \big) \right) \leftarrow R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}). \tag{6.9}$$

Queries (6.8) and (6.9) are of the same form as the aggregate queries introduced in

Section 5.3, which means that their computation can be factorized and computed in time given by Theorem 5.15. In particular, following Proposition 5.20 the batch of aggregate queries can be computed in time sublinear in $|D|$, i.e., the time it takes to output the feature extraction query. Batch gradient descent optimization, however, requires that the queries are recomputed for each iteration of the algorithm (cf. Section 6.2).

We next present an alternative approach to computing the data dependent computation of $\frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_k}$, where each batch gradient descent iteration does not require any computation over the input data. This approach applies distributivity of product over summation operators to rewrite the partial derivative as follows:

$$\forall k \in [n] \; : \; \frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_k} = \sum_{\boldsymbol{x} \in D} \Big(\sum_{j \in [n+1]} \boldsymbol{\theta}_j^\top \boldsymbol{x}_j \Big) \boldsymbol{x}_k^\top + \lambda \boldsymbol{\theta}_k \tag{6.10}$$

$$= \sum_{j \in [n+1]} \boldsymbol{\theta}_j^\top \sum_{\boldsymbol{x} \in D} \boldsymbol{x}_j \boldsymbol{x}_k^\top + \lambda \boldsymbol{\theta}_k \tag{6.11}$$

$$= \sum_{j \in [n+1]} \boldsymbol{\theta}_j^\top \boldsymbol{\sigma}_{jk} + \lambda \boldsymbol{\theta}_k, \tag{6.12}$$

where $\boldsymbol{\sigma}_{jk} = \sum_{\boldsymbol{x} \in D} \boldsymbol{x}_j \boldsymbol{x}_k^\top$. We use the $\boldsymbol{\Sigma} = (\boldsymbol{\sigma}_{ij})_{i,j \in [n+1]}$ to denote the block matrix, for which each (i, j) entry is defined by the $\boldsymbol{\sigma}_{ij}$ matrices.

Based on Equation 6.12, we can also define the entire gradient vector in terms of $\boldsymbol{\Sigma}$:

$$\boldsymbol{\nabla} J(\boldsymbol{\theta}) = \boldsymbol{\theta}^\top \Big(\sum_{\boldsymbol{x} \in D} \boldsymbol{x} \boldsymbol{x}^\top \Big) + \lambda \boldsymbol{\theta} \tag{6.13}$$

$$= \boldsymbol{\theta}^\top \boldsymbol{\Sigma} + \lambda \boldsymbol{\theta}. \tag{6.14}$$

The matrix $\boldsymbol{\Sigma}$ represents the non-centered covariance matrix [92], called the covar matrix hereafter. For each (j, k)-entry in the covar matrix, there exists a relational representation of $\boldsymbol{\sigma}_{jk}$ which can be computed with one aggregate query of the form (4.1). The exact form of the query depends on the type of X_j and X_k. Recall that the training dataset D is defined by a feature extraction query Q over a database instance with relations $R_1, \ldots, R_m$.

If X_j and X_k are continuous variables, then $\boldsymbol{x}_j$ and $\boldsymbol{x}_k$ are both scalars. In this case, $\boldsymbol{\sigma}_{ij} = \sum_{\boldsymbol{X} \in D} x_j \cdot x_k$, which can be computed with a simple `SUM` aggregate query:

$$\mathsf{Covar}_{j,k}(\texttt{SUM}(X_j \cdot X_k)) \leftarrow R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}). \tag{6.15}$$

If X_j is continuous and X_k categorical, then $\boldsymbol{x}_j$ is a scalar, $\boldsymbol{x}_k$ encodes the indicator vector $(\mathbf{1}_{X_k = v})_{v \in \mathsf{Dom}(X_k)}$. In this case, $\boldsymbol{\sigma}_{ij} = \sum_{\boldsymbol{X} \in D} x_j \cdot \boldsymbol{x}_k$ computes a vector with one entry for each $v \in \mathsf{Dom}(X_k)$. For each $v \in \mathsf{Dom}(X_k)$, the corresponding entry in $\boldsymbol{\sigma}_{ij}$ represents the summation over X_j for all tuples $(\boldsymbol{x}, y) \in D$ where $x_k = v$. This vector has a relational

representation which can be computed as an aggregate query with X_k as group-by variable:

$$\mathsf{Covar}_{j,k}(X_k, \texttt{SUM}(X_j)) \leftarrow R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}). \tag{6.16}$$

If both X_j and X_k are categorical, then $\boldsymbol{x}_j$ and $\boldsymbol{x}_k$ encode indicator vectors. In this case, $\boldsymbol{\sigma}_{ij}$ encodes a matrix, where each entry (x_j, x_k) gives the number of tuples $(\boldsymbol{x}, y) \in D$ where $X_j = x_j$ and $X_k = x_k$. Note that many entries in $\boldsymbol{\sigma}_{ij}$ can be zero. A sparse, relational encoding of $\boldsymbol{\sigma}_{jk}$ is given by an aggregate query with group-by variables X_j and X_k:

$$\mathsf{Covar}_{j,k}(X_j, X_k, \texttt{SUM(1)}) \leftarrow R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}). \tag{6.17}$$

Example 6.6. Consider the linear regression model from Example 6.4. The update rule for parameter θ_{price} would require the computation of the following queries, each of which correspond to an entry in the covar matrix:

$$\begin{array}{ll}
\mathsf{Covar}_{\mathsf{price},\mathsf{intrcpt}}(\texttt{SUM(}\mathsf{price}\texttt{)}) & \mathsf{Covar}_{\mathsf{price},\mathsf{price}}(\texttt{SUM(}\mathsf{price} \cdot \mathsf{price}\texttt{)}) \\
\mathsf{Covar}_{\mathsf{price},\mathsf{size}}(\texttt{SUM(}\mathsf{price} \cdot \mathsf{size}\texttt{)}) & \mathsf{Covar}_{\mathsf{price},\mathsf{city}}(\mathsf{city}, \texttt{SUM(}\mathsf{price}\texttt{)}) \\
\mathsf{Covar}_{\mathsf{price},\mathsf{color}}(\mathsf{color}, \texttt{SUM(}\mathsf{price}\texttt{)}) & \mathsf{Covar}_{\mathsf{price},\mathsf{units}}(\texttt{SUM(}\mathsf{price} \cdot \mathsf{units}\texttt{)})
\end{array}$$

All queries are computed over the same join body: $\mathsf{Sales}, \mathsf{Items}, \mathsf{Stores}$. □

Following the computation of the covar matrix, we compute the partial derivatives $\frac{\partial}{\partial \boldsymbol{\theta}_k} J(\boldsymbol{\theta})$ by multiplying each $(\boldsymbol{\sigma}_{kj})_{j \in [n]}$ with the corresponding parameter $\boldsymbol{\theta}_j$ and then summing over their results. This can be encoded as an aggregate query over the join of the corresponding Covar aggregate and a relational encoding of the parameter vector $\boldsymbol{\theta}_j$. For a continuous feature X_j, the relational encoding of $\boldsymbol{\theta}_j$ is a unary relation with a single value that stores the paramater θ_{X_j}. For a categorical feature X_j, the relational encoding of $\boldsymbol{\theta}_j$ is a binary relation $\boldsymbol{\theta}_j(X_j, p_j)$, which has one tuple (x_j, p) for each value $x_j \in \mathsf{Dom}(X_j)$ where p represents the parameter θ_{x_j} in $\boldsymbol{\theta}_j$ that corresponds to x_j.

Consider the case where X_j and X_k are categorical variables. The query that encodes the product of $\boldsymbol{\sigma}_{kj}$ and $\boldsymbol{\theta}_j$ is then given by:

$$Q(X_k, \texttt{SUM(}P \cdot C\texttt{)}) = \mathsf{Covar}_{ij}(X_j, X_k, C), \boldsymbol{\theta}_j(X_j, P). \tag{6.18}$$

Similarly, if X_k is categorical, but X_j is continuous, we can encode the product as follows:

$$Q(X_k, \texttt{SUM(}P \cdot C\texttt{)}) = \mathsf{Covar}_{ij}(X_k, C), \boldsymbol{\theta}_j(P). \tag{6.19}$$

If variables X_j and X_k are both continuous, then the product of $\boldsymbol{\sigma}_{kj}$ and $\boldsymbol{\theta}_j$ is a simple product of two scalars.

Example 6.7. Consider the linear regression model from Example 6.4, and the Covar

queries from Example 6.6. To compute the data dependent part of the partial derivative $\frac{\partial}{\partial \boldsymbol{\theta}_{\mathsf{price}}} J(\boldsymbol{\theta})$, we first compute the following queries for the categorical variables city and color:

$$Q_{\mathsf{city}}(\mathtt{SUM}(c \cdot p)) = \mathsf{Covar}_{\mathsf{price,city}}(\mathsf{city}, c), \boldsymbol{\theta}_{\mathsf{city}}(\mathsf{city}, p)$$
$$Q_{\mathsf{color}}(\mathtt{SUM}(c \cdot p)) = \mathsf{Covar}_{\mathsf{price,color}}(\mathsf{color}, c), \boldsymbol{\theta}_{\mathsf{color}}(\mathsf{color}, p)$$

where $\mathsf{Covar}_{\mathsf{price,city}}(\mathsf{city}, c)$ and $\boldsymbol{\theta}_{\mathsf{city}}(\mathsf{city}, p)$ join on city, and $\mathsf{Covar}_{\mathsf{price,color}}(\mathsf{color}, c)$ and $\boldsymbol{\theta}_{\mathsf{color}}(\mathsf{color}, p)$ join on color.

The remaining parameters and Covar queries encode scalars. The partial derivate $\frac{\partial}{\partial \boldsymbol{\theta}_{\mathsf{price}}} J(\boldsymbol{\theta})$ is thus given by:

$$\begin{aligned}\mathsf{Partial}_{\mathsf{price}}(\mathtt{SUM}(p_1 \cdot c_1 + p_2 \cdot c_2 + p_3 \cdot c_3 + c_4 + c_5 - c_6)) \leftarrow \\ \mathsf{Covar}_{\mathsf{price,intrcpt}}(c_1), \mathsf{Covar}_{\mathsf{price,price}}(c_2), \mathsf{Covar}_{\mathsf{price,size}}(c_3), \mathsf{Covar}_{\mathsf{price,units}}(c_6), \\ Q_{\mathsf{city}}(c_4), Q_{\mathsf{color}}(c_5), \boldsymbol{\theta}_{\mathsf{intrcpt}}(p_1), \boldsymbol{\theta}_{\mathsf{price}}(p_2), \boldsymbol{\theta}_{\mathsf{size}}(p_3)\end{aligned}$$

Note that the c_6 is multiplied by -1, as it encodes an aggregate over the over label Y. □

The rewriting of the data dependent computation of the gradient into Covar queries has several benefits:

1. The covar matrix does not depend on the parameters $\boldsymbol{\theta}$, and can be computed once for all batch gradient descent iterations. As a result, each batch gradient descent iteration is independent of $|D|$.

2. By capturing categorical variables as group-by variables in the Covar queries, we can compute the model directly over the raw input database and without one-hot encoding the input data.

3. The covar matrix commutes with projection. This means that once we compute the covar matrix, we can reuse it to learn any model defined by a subset of the original features. This model would be computed over the projection of the covar matrix onto this subset of features, and without any computation over the input data. This property is desirable for model selection, which aims to find a subset of features that best predict the label. We provide additional details on model selection in Section 6.6.

In the following, we assume that we learn models with square loss functions using the covar matrix instead of computing the partial derivatives directly.

Complexity analysis

We next give runtime bounds for the learning of linear regression models over feature extraction queries. We assume that the model is learned over the covar matrix. We use

$|\boldsymbol{\sigma}_{ij}|$ to denote the number of tuples in the result of the query that represents the (i, j)-entry of the covar matrix.

Theorem 6.8. *Let Q be a feature extraction query with variables $X_1, \ldots, X_n$ that is computed over a database instance I, where all relations in I have size at most N. Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ denote the query hypergraph of Q.*

Any linear regression model $LR(\boldsymbol{x})$ can be learned over Q with ξ batch gradient descent iterations in time:

$$O\left(|\mathcal{V}|^2 \cdot |\mathcal{E}| \cdot n^2 \cdot N^{\mathsf{fhtw}(\mathcal{H})+1} \cdot \log N + \xi \cdot \sum_{i,j \in [n]} |\boldsymbol{\sigma}_{ij}|\right). \tag{6.20}$$

Proof. Consider the first component of (6.20), which captures the time it takes to compute the matrix $\boldsymbol{\Sigma}$.

We have shown above that each entry $\boldsymbol{\sigma}_{ij} \in \boldsymbol{\Sigma}$ can be rewritten as one aggregate query of the form (6.15), (6.16), or (6.17). Each of these queries can have at most two free variables, and is of the same form as the aggregate query (4.1). The complexity for computing all entries in $\boldsymbol{\Sigma}$ directly follows from Theorem 5.15, which states that each of these queries can be computed in time $O\left(|\mathcal{V}|^2 \cdot |\mathcal{E}| \cdot (N^{\mathsf{fhtw}_F(\mathcal{H})} + |\boldsymbol{\sigma}_{ij}|) \cdot \log N\right)$, as well as Proposition 5.13, which states that $\mathsf{fhtw}_F(\mathcal{H}) < \mathsf{fhtw}(\mathcal{H}) + 1$ (since there are at most two free variables). In addition, Proposition 5.20 states that $|\boldsymbol{\sigma}_{ij}| \leq N^2 \leq N^{\mathsf{fhtw}(\mathcal{H})+1}$. Thus, the time it takes to compute the output is always upper bounded by the time it takes to evaluate the query, and we can hide the factor $|\boldsymbol{\sigma}_{ij}|$ in the first part of (6.20).

The second component of (6.20) captures the runtime for batch gradient descent optimization over the precomputed matrix $\boldsymbol{\Sigma}$. The complexity follows from the fact that each iteration $\ell \in [\xi]$ requires the computation over the gradient vector, which can be computed in one pass over $\boldsymbol{\Sigma}$ and in time $O(\sum_{i,j \in [n]} |\boldsymbol{\sigma}_{ij}|)$. □

Based on Proposition 5.20, our solution for learning linear regression models can be computed in strictly sub-linear time in the output size of the underlying feature extraction query Q. If all features in D are continuous, then $\mathsf{fhtw}_F(\mathcal{H}) = \mathsf{fhtw}(\mathcal{H})$ (cf. Section 5.2) and each $\boldsymbol{\sigma}_{ij}$ represents a scalar. Then, the overall runtime for learning linear regression models over Q becomes:

$$O(|\mathcal{V}|^2 \cdot |\mathcal{E}| \cdot n^2 \cdot N^{\mathsf{fhtw}(\mathcal{H})} \cdot \log N + \xi \cdot n^2).$$

6.4 Polynomial Regression

A degree-d polynomial regression model extends the linear regression model from Equation (6.4) with additional terms that are defined by monomials of input features up to degree d. These additional features are commonly called *feature interactions*, because they

model the interaction effect of the features on the label Y. The model multiplies each feature interaction term by a corresponding unique parameter.

Example 6.9. Consider the retailer scenario from Example 6.4, which aims to predict future sales, based on the city and size of the store, and the color and price of the product.

The parameters of a linear regression model capture linear effects between the feature and the label units sold. For instance, the term $\theta_{\mathsf{oxford}} x_{\mathsf{oxford}}$ can encode that the number of units sold at a store in Oxford is relatively low. The reason could be that Oxford is a student city and students typically spend less.

The relationship between Oxford and the number of units sold, however, may not always hold. For instance, it is reasonable to assume that the number of units sold for *blue* products is relatively *high* at Oxford stores, because blue is the color of Oxford University and tourists like to buy Oxford merchandise. This interaction effect between the color blue and city Oxford can be modeled by extending the LR model with the feature interaction term $\theta_{\mathsf{blue,oxford}} \mathsf{blue} \cdot \mathsf{oxford}$. □

Let D be a training dataset with variables $X_1, \ldots, X_n$ and label Y that is the result of feature extraction query Q over a database with relations $R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m})$. As in Section 6.3, for each tuple $(\boldsymbol{x}, y) \in D$, the vector $\boldsymbol{x}$ denotes the feature vector where categorical features are one-hot encoded indicator vectors.

To define a polynomial regression model of arbitrary degree d, we define a feature map $h : R^n \to R^\ell$, which transforms the raw feature vector $\boldsymbol{x}$ into an ℓ-dimensional vector of monomial features $h(\boldsymbol{x}) = (h_j(\boldsymbol{x}))_{j \in [\ell]}$. Each component h_j is a multivariate *monomial* designed to capture the *interactions* among dimensions of the input $\boldsymbol{x}$:

$$h_j(\boldsymbol{x}) = \bigotimes_{i \in [n]} \boldsymbol{x}_i^{a_j(i)}. \tag{6.21}$$

The degree $a_j(i)$ gives the level of participation of input dimension i in the j-th monomial.

For each $j \in [\ell]$, the set $V_j = \{k \in [n] \mid a_j(k) > 0\}$ consists of variables that participate in the interaction captured by the monomial h_j. Let $C \subseteq [n]$ denote the set of categorical variables, and $C_j = C \cap V_j$ the subset of categorical variables in V_j. Then, h_j represents $\prod_{k \in C_j} |\mathsf{Dom}(X_k)|$ many monomials, one for each combination of the categories.

Due to one-hot encoding, each element in the vector $\boldsymbol{x}_f$ for a categorical variable X_f is either 0 or 1, and thus $\boldsymbol{x}_f^{a_j(f)} = \boldsymbol{x}_f$ for all $a_j(f) > 0$. Hence, h_j can be simplified as follows:

$$h_j(\boldsymbol{x}) = \prod_{i \in V_j \setminus C_j} x_i^{a_j(i)} \bigotimes_{k \in C_j} \boldsymbol{x}_k. \tag{6.22}$$

Note that we use x_i instead of boldface $\boldsymbol{x}_i$ since each variable $i \in V_j \setminus C_j$ is continuous.

The parameters of a polynomial regression model can be encoded as a vector $\boldsymbol{\theta} = (\boldsymbol{\theta}_j)_{j \in [\ell]}$, where each component $\boldsymbol{\theta}_j$ represents a tensor of scalar parameters which has the

same dimensions as h_j.

We can now define the degree-d polynomial regression model as follows:

$$\mathsf{PR}_d(\boldsymbol{x}) = \sum_{i \in [\ell]} \langle \boldsymbol{\theta}_i, h(\boldsymbol{x}_i) \rangle = \langle \boldsymbol{\theta}, h(\boldsymbol{x}) \rangle \,, \tag{6.23}$$

where $\langle \boldsymbol{\theta}, h(\boldsymbol{x}) \rangle$ encodes the Frobenius inner product.

We next exemplify the terms in a polynomial regression model of degree 2.

Example 6.10. Consider the linear regression model from Example 6.4. A polynomial regression model of degree-2 extends this model with all pairwise monomials over the features.

For instance, for price and size the PR_2 model would contain the term $\theta_{(\mathsf{price},\mathsf{size})}\mathsf{price}{\cdot}\mathsf{size}$, where $\theta_{(\mathsf{price},\mathsf{size})}$ is the scalar parameter for this interaction.

Now, consider variables city and color. Then, the PR_2 model includes one term for each possible combination of city and color, including $\theta_{\mathsf{oxford},\mathsf{red}} x_{\mathsf{oxford}} x_{\mathsf{red}}$, $\theta_{\mathsf{london},\mathsf{blue}} x_{\mathsf{london}} x_{\mathsf{blue}}$, etc. All these terms are captured by the Frobenius inner product $\left\langle \boldsymbol{\theta}_{(\mathsf{city},\mathsf{color})}, \boldsymbol{x}_{\mathsf{city}} \otimes \boldsymbol{x}_{\mathsf{color}} \right\rangle$. The component $\boldsymbol{\theta}_{(\mathsf{city},\mathsf{color})}$ is a matrix with one parameter for each pair of $(\mathsf{city}, \mathsf{color})$. □

Learning polynomial regression models

We learn the parameters of the polynomial regression model using batch gradient descent optimization as shown in Algorithm 6.1. The goal is to minimize the objective function $J(\boldsymbol{\theta})$ with square-loss and ℓ_2-regularization:

$$J(\boldsymbol{\theta}) = \frac{1}{2|D|} \sum_{\boldsymbol{x} \in D} \big(\langle \boldsymbol{\theta}, h(\boldsymbol{x}) \rangle - y \big)^2 + \frac{\lambda}{2} \left\| \boldsymbol{\theta} \right\|_2^2 . \tag{6.24}$$

As for linear regression models, we extend, without loss of generality, the feature map $h(\boldsymbol{x})$ with an additional term $h_{\ell+1}(\boldsymbol{x}) = y$, and define the corresponding parameter $\boldsymbol{\theta}_{\ell+1}$ which always takes value -1. Then, we can treat the features and label uniformly, and simplify the expression of the objective function as follows:

$$\begin{aligned}
J(\boldsymbol{\theta}) &= \frac{1}{2|D|} \sum_{\boldsymbol{x} \in D} \big(\langle \boldsymbol{\theta}, h(\boldsymbol{x}) \rangle \big)^2 + \frac{\lambda}{2} \left\| \boldsymbol{\theta} \right\|_2^2 && (6.25) \\
&= \frac{1}{2|D|} \sum_{\boldsymbol{x} \in D} \boldsymbol{\theta}^\top h(\boldsymbol{x}) h(\boldsymbol{x})^\top \boldsymbol{\theta} + \frac{\lambda}{2} \left\| \boldsymbol{\theta} \right\|_2^2 && (6.26) \\
&= \frac{1}{2|D|} \boldsymbol{\theta}^\top \big(\sum_{\boldsymbol{x} \in D} h(\boldsymbol{x}) h(\boldsymbol{x})^\top \big) \boldsymbol{\theta} + \frac{\lambda}{2} \left\| \boldsymbol{\theta} \right\|_2^2 && (6.27) \\
&= \frac{1}{2|D|} \boldsymbol{\theta}^\top \boldsymbol{\Sigma} \boldsymbol{\theta} + \frac{\lambda}{2} \left\| \boldsymbol{\theta} \right\|_2^2 . && (6.28)
\end{aligned}$$

Similarly, we derive the gradient $\boldsymbol{\nabla} J(\boldsymbol{\theta})$ straightforwardly from Equation (6.28) using

the chain rule:

$$\nabla J(\boldsymbol{\theta}) = \boldsymbol{\Sigma}\boldsymbol{\theta} + \lambda\boldsymbol{\theta}. \tag{6.29}$$

In Equations (6.28) and (6.29), $\boldsymbol{\Sigma}$ is a $p \times p$ block matrix, where each entry in $\boldsymbol{\sigma}_{ij} \in \boldsymbol{\Sigma}$ is a tensor defined by the outer product of h_i and h_j:

$$\boldsymbol{\sigma}_{ij} = \sum_{(\boldsymbol{x},y)\in D} h_i(\boldsymbol{x}) h_j(\boldsymbol{x})^\top. \tag{6.30}$$

More specifically, let C_i and C_j be the set of categorical variables for h_i and h_j. Then, $\boldsymbol{\sigma}_{ij}$ is a tensor of dimension $\prod_{j\in C_i} |\mathsf{Dom}(X_j)| \times \prod_{j\in C_j} |\mathsf{Dom}(X_j)|$. The entries $\boldsymbol{\sigma}_{ij}$ generalize the Covar aggregates for linear regression in Section 6.3 to higher dimensions. Let $V_{ij} = (V_i \cup V_j)$ be the set of all variables that occur in h_i and h_j, and $C_{ij} = C_i \cup C_j$ denote the set of categorical variables in h_i and h_j. We further define $a_{ij}(k) = a_i(k) + a_j(k)$, $\forall k \in V_{ij}$. Then, we can compute a sparse, relational representation of $\boldsymbol{\sigma}_{ij}$ with the following query:

$$\mathsf{Covar}_{ij}(C_{ij}, \texttt{SUM}(\prod\nolimits_{k\in V_{ij}\setminus C_{ij}} X_k^{a_{ij}(k)})) = R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}) \tag{6.31}$$

Example 6.11. Consider the tensor $\boldsymbol{\sigma}_{ij} \in \boldsymbol{\Sigma}$ where $i = (\mathsf{city}, \mathsf{country})$ and $j = (\mathsf{color}, \mathsf{store})$:

$$\boldsymbol{\sigma}_{ij} = \sum_{\boldsymbol{x}\in D} (\boldsymbol{x}_{\mathsf{city}} \otimes \boldsymbol{x}_{\mathsf{country}})(\boldsymbol{x}_{\mathsf{color}} \otimes \boldsymbol{x}_{\mathsf{store}})^\top.$$

Following the one-hot encoding of categorical variables, $\boldsymbol{x}_{\mathsf{city}}$, $\boldsymbol{x}_{\mathsf{country}}$, $\boldsymbol{x}_{\mathsf{color}}$, and $\boldsymbol{x}_{\mathsf{store}}$ are indicator vectors. The dimensionality of $\boldsymbol{\sigma}_{ij}$ is thus given by:

$$|\mathsf{Dom}(\mathsf{city})| \cdot |\mathsf{Dom}(\mathsf{country})| \times |\mathsf{Dom}(\mathsf{color})| \cdot |\mathsf{Dom}(\mathsf{store})|.$$

The tensor $\boldsymbol{\sigma}_{ij}$ has the following straightforward interpretation: every tensor entry $(x_{\mathsf{city}}, x_{\mathsf{country}}, x_{\mathsf{color}}, x_{\mathsf{store}})$ represents the number of data points $(\boldsymbol{x}, y) \in D$ where this particular combination of values for city, $\mathsf{country}$, color, and store appear together. Most of these entries are 0. For example, if x_{store} is not located in x_{city}, or x_{city} is not in x_{country}, then the count is zero. In fact, since store functionally determines city and $\mathsf{country}$, the number of non-zero entries in the tensor is upper bounded by $|\mathsf{Dom}(\mathsf{color})| \times |\mathsf{Dom}(\mathsf{store})|$.

We can compute a sparse representation of all non-zero entries in $\boldsymbol{\sigma}_{ij}$ with the query:

$$\mathsf{Covar}(\mathsf{city}, \mathsf{country}, \mathsf{store}, \mathsf{color}, \texttt{SUM(1)}) \leftarrow \mathsf{Items}(\mathsf{sku}, \mathsf{color}, \mathsf{price}), \mathsf{Stores}(\mathsf{store}, \mathsf{city}, \mathsf{size}), \mathsf{Sales}(\mathsf{sku}, \mathsf{store}, \mathsf{day}, \mathsf{units\ sold})$$

We can also succinctly represent tensors that are composed of continuous and categorical

variables. Consider the tensor $\boldsymbol{\sigma}_{ik}$ where $i = (\mathsf{city}, \mathsf{country})$ and $k = (\mathsf{color}, \mathsf{price})$:

$$\sum_{\boldsymbol{x} \in D} (\boldsymbol{x}_{\mathsf{city}} \otimes \boldsymbol{x}_{\mathsf{country}})(x_{\mathsf{price}} \cdot \boldsymbol{x}_{\mathsf{color}})^\top.$$

We can compute this tensor with the following aggregate query:

$$\mathsf{Covar}(\mathsf{city}, \mathsf{country}, \mathsf{color}, \texttt{SUM(price)}) \leftarrow \mathsf{Items}(\mathsf{sku}, \mathsf{color}, \mathsf{price}), \mathsf{Stores}(\mathsf{store}, \mathsf{city}, \mathsf{size}), \mathsf{Sales}(\mathsf{sku}, \mathsf{store}, \mathsf{day}, \mathsf{units\ sold})$$

□

An additional benefit of representing the tensors entries of $\boldsymbol{\Sigma}$ as relations is that many $\boldsymbol{\sigma}_{ij}$ tensors have an identical relational representation. In this case, we can compute the aggregate query that defines these tensors only once, and reuse the same result for different entries in $\boldsymbol{\Sigma}$.

Example 6.12. Consider the tensor $\boldsymbol{\sigma}_{ij}$ where $i = (\mathsf{city}, \mathsf{country})$ and $j = (\mathsf{city}, \mathsf{store})$ and another (different) tensor $\boldsymbol{\sigma}_{k,\ell}$ where $k = (\mathsf{city}, \mathsf{country})$ and $\ell = (\mathsf{country}, \mathsf{store})$. These two tensors have an identical sparse, relational representation, which can be computed with the following query:

$$\mathsf{Covar}(\mathsf{city}, \mathsf{country}, \mathsf{store}, \texttt{SUM(1)}) \leftarrow \mathsf{Items}(\mathsf{sku}, \mathsf{color}, \mathsf{price}), \mathsf{Stores}(\mathsf{store}, \mathsf{city}, \mathsf{size}), \mathsf{Sales}(\mathsf{sku}, \mathsf{store}, \mathsf{day}, \mathsf{units\ sold})$$

In fact, Covar is the sparse representation for any tensor $\boldsymbol{\sigma}_{ij}$ where

$$(i, j) \in \{((\mathsf{city}, \mathsf{country}), \mathsf{store}), (\mathsf{city}, (\mathsf{country}, \mathsf{store})), ((\mathsf{city}, \mathsf{store}), (\mathsf{country}, \mathsf{store})), \ldots\}.$$

This is because city, store, and $\mathsf{country}$ are categorical variables and taking any power of the binary values in their indicator vectors does not change these values. Furthermore, any of the variables can be in i and/or j. □

Finally, we show how to compute the partial derivatives of $J(\boldsymbol{\theta})$ over the precomputed $\boldsymbol{\Sigma}$ matrix. Equation (6.29) shows that the partial derivative $\frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_k}$ with respect to $\boldsymbol{\theta}_k$ is given by:

$$\frac{\partial}{\partial \boldsymbol{\theta}_k} J(\boldsymbol{\theta}) = \sum_{i \in [\ell+1]} \boldsymbol{\sigma}_{ki} \boldsymbol{\theta}_i + \lambda \boldsymbol{\theta}_k. \tag{6.32}$$

Note that we would only need to compute the partial derivatives for each $j \in [\ell]$, as we never update parameter $\boldsymbol{\theta}_{\ell+1}$ which is fixed to -1.

Equation (6.32) can be computed as an aggregate query of the form (4.1), where the join is defined by the precomputed Covar queries, which represent the tensors $\boldsymbol{\sigma}_{ij}$ and a relational encoding of the parameter tensors $\boldsymbol{\theta}_j$.

Example 6.13. Consider the tensor $\boldsymbol{\sigma}_{ij}$ where $i = (\mathsf{city}, \mathsf{country})$ and $j = (\mathsf{city}, \mathsf{store})$ from Example 6.12. The partial derivative for $\boldsymbol{\theta}_i$ as defined in Equation (6.29) requires the computation of $\boldsymbol{\sigma}_{ij}\boldsymbol{\theta}_j$.

We encode the parameter tensor $\boldsymbol{\theta}_{(\mathsf{city},\mathsf{store})}$ as a ternary relation $\boldsymbol{\theta}_{(\mathsf{city},\mathsf{store})}(\mathsf{city}, \mathsf{store}, p)$, which stores for each pair $(x_{\mathsf{city}}, x_{\mathsf{store}})$ the corresponding parameter $\theta_{(x_{\mathsf{city}},x_{\mathsf{store}})}$. The number of entries in the relation is given by the number of pairs $(\mathsf{city}, \mathsf{store})$ that appear together in some tuple in the training dataset. This number can be much less than the product of the numbers of countries and of colors in the input database.

The product $\boldsymbol{\sigma}_{ij}\boldsymbol{\theta}_j$ can then be computed with the following query:

$$Q(\mathsf{city}, \mathsf{country}, \texttt{SUM}(c * p)) = \mathsf{Covar}_{(\mathsf{city},\mathsf{country},\mathsf{store})}(\mathsf{city}, \mathsf{country}, \mathsf{store}, c), \boldsymbol{\theta}_{(\mathsf{city},\mathsf{store})}(\mathsf{city}, \mathsf{store}, p).$$

□

Complexity analysis

We next give the runtime complexity for learning degree-d polynomial regression models over feature extraction queries. We assume that the model is learned over the covar matrix $\boldsymbol{\Sigma} = (\boldsymbol{\sigma}_{ij})_{i,j\in[\ell]}$. Let $C_{ij} = C_i \cup C_j$ denote the group-by variables, and $|\boldsymbol{\sigma}_{ij}|$ the number of tuples, in the query of form (6.31) that represents the (i, j)-entry of $\boldsymbol{\Sigma}$.

Theorem 6.14. *Let Q be a feature extraction query over a database instance I where all relations have size at most N. Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ denote the query hypergraph of Q.*

Any degree-d polynomial regression model $\mathsf{PR}_d(\boldsymbol{x})$ can be learned over Q with ξ batch gradient descent iterations in time:

$$O\left(|\mathcal{V}|^2 \cdot |\mathcal{E}| \cdot \sum_{i,j\in[\ell]} (N^{\mathsf{fhtw}_{C_{ij}}(\mathcal{H})} + |\boldsymbol{\sigma}_{ij}|) \cdot \log N + \xi \cdot \sum_{i,j\in[\ell]} |\boldsymbol{\sigma}_{ij}|\right). \tag{6.33}$$

where ℓ is the dimension of the feature mapping vector $h(\boldsymbol{x})$.

Proof. The first part of (6.33) captures the computation of $\boldsymbol{\Sigma}$ over Q. Recall that each entry $\boldsymbol{\sigma}_{ij} \in \boldsymbol{\Sigma}$ can be computed with aggregate query (6.31). The complexity for each query (6.31) directly follows from Theorem 5.15, which states that each of these queries can be computed in time $O\left(|\mathcal{V}|^2 \cdot |\mathcal{E}| \cdot (N^{\mathsf{fhtw}_{C_{ij}}(\boldsymbol{\sigma}_{ij})} + |\boldsymbol{\sigma}_{ij}|) \cdot \log N\right)$.

The second term in (6.33) captures the BGD optimization, which follows from the fact that each BGD iteration can be computed in one pass over $\boldsymbol{\Sigma}$. □

From Proposition 5.20, it follows that the degree-d polynomial regression model can thus be computed in time sublinear in $|Q|$ for infinitely many features extraction queries Q.

In case all variables in D are continuous, then $\mathsf{fhtw}_{C_{ij}}(\mathcal{H}) = \mathsf{fhtw}(\mathcal{H})$ and the overall runtime for computing the matrix $\boldsymbol{\Sigma}$ becomes $O(|\mathcal{V}|^2 \cdot |\mathcal{E}| \cdot \ell^2 \cdot N^{\mathsf{fhtw}(\mathcal{H})} \cdot \log N)$.

6.5 Factorization Machines

Factorization machines are machine learning models that have predominantly been used in recommender system scenarios, where the model seeks to predict "preferences" or "ratings" that users assign to a set of products [114, 116, 115].

In such scenarios, the input data is typically represented as a matrix with one dimension for users and the other for products. The values in the matrix correspond to ratings that users gave to products. A conventional recommender systems model, the model is typically a specialized technique based on matrix factorization algorithms, which decompose this matrix into latent factors for each user and product. These latent factors are then used to predict the rating a user would give to a product that this user had not rated before. In contrast, factorization machines treat the recommendation problem as a regression problem, where the latent factors are encoded in the parameters of the model. This means that factorization machines are also general purpose predictors, which can thus be used in many other use cases, including the retail forecasting scenario motivated in Section 6.3. An additional advantage is that factorization machines can be easily extended with additional features that provide characteristics for users and products.

The structure of a factorization machine is similar to a polynomial regression model, as both models extend linear regression with feature interaction up to degree d. The main differences between factorization machines and polynomial regression models are: (1) factorization machines factorize the model parameters to better capture data correlations, and (2) a feature interaction monomial in a factorization machine can have degree at most one (e.g., you cannot have terms like x_i^2 or $x_i^3 \cdot \boldsymbol{x}_j$). The latter is because the parameter factorization prevents factorization machines from modeling negative interaction effects of features interaction terms with higher degree monomials.

Example 6.15. Consider the terms of the PR_2 model from Example 6.10. For the interaction between price and size, the degree 2, rank $\mathcal{R}$ factorization machine contains the term:

$$\sum_{r \in [\mathcal{R}]} \theta_{\mathsf{price},r} \cdot \theta_{\mathsf{size},r} \cdot \mathsf{price} \cdot \mathsf{size}$$

where $\sum_{r \in [\mathcal{R}]} \theta_{\mathsf{price},r} \cdot \theta_{\mathsf{size},r}$ factorizes the parameter $\theta_{(\mathsf{price},\mathsf{size})}$ from the PR_2 model.

For the interaction term between city and color, the degree-2, rank-$\mathcal{R}$ factorization machine would contain the term:

$$\left\langle \sum_{r \in [\mathcal{R}]} \boldsymbol{\theta}_{\mathsf{city},r} \otimes \boldsymbol{\theta}_{\mathsf{color},r} \ , \ \boldsymbol{x}_{\mathsf{city}} \otimes \boldsymbol{x}_{\mathsf{color}} \right\rangle \tag{6.34}$$

where $\sum_{r \in [\mathcal{R}]} \boldsymbol{\theta}_{\mathsf{city},r} \otimes \boldsymbol{\theta}_{\mathsf{color},r}$ represents a decomposition of the matrix of parameters $\boldsymbol{\theta}_{(\mathsf{city},\mathsf{color})}$

that is learned by the PR_2 model. □

The parameter factorization is particularly useful for recommender systems scenarios, where the feature space is typically very sparse as a result of the one-hot encoding of categorical variables.

Example 6.16. In a simplified recommender systems scenario, the model is learned over a training dataset with variables: user, item, and rating. The variables user and item are identifiers for a given user and respectively product, and rating is a scalar.

The goal is to learn a model over the one-hot encoded features for user and item that can predict the rating a user gives to the product. Note that, due to the one-hot encoding of user and item, the model has one feature for each user and product, and the feature space for the model is very sparse.

A polynomial regression model would contain the term:

$$\left\langle \boldsymbol{\theta}_{(\mathsf{user},\mathsf{item})}, \boldsymbol{x}_{\mathsf{user}} \otimes \boldsymbol{x}_{\mathsf{item}} \right\rangle$$

which captures all pairwise interactions between users and items, and assigns a specific parameter to each such interaction. It is likely, however, that each user x_{user} rates a given item x_{item} only once, such that the training dataset has at most one tuple that contains the pair $(x_{\mathsf{user}}, x_{\mathsf{item}})$. This makes learning the PR_2 model infeasible, because there are not enough data points to learn the parameter $\theta_{(x_{\mathsf{user}},x_{\mathsf{item}})}$ for this particular feature interaction.

Parameter factorization in factorization machines mitigates this issue, because each factorized parameter $\theta_{(x_{\mathsf{user}},r)}$ for each $r \in [\mathcal{R}]$ is involved in the interaction terms with all items that x_{user} has rated. Therefore, there are multiple tuples in the training dataset over which this particular parameter is learned.

Once learned, the factorized parameters $\boldsymbol{\theta}_{\mathsf{user},r}$ and $\boldsymbol{\theta}_{\mathsf{item},r}$ for each $r \in [\mathcal{R}]$ capture latent factors that describe each user and respectively item. These factors are similar to latent factors represented by the low-dimensional matrices that are derived by collaborative filtering with matrix factorization. □

We next present a uniform encoding for factorization machines of arbitrary degree d. Let D denote a training dataset that is defined by a feature extraction query Q with variables $\boldsymbol{X} = \{X_1, \ldots, X_n\}$ and label Y that is computed over a database with relations $R_1, \ldots, R_m$.

For the input variables $\boldsymbol{X}$, the feature interaction terms of a degree-d factorization machine are identical to those of a degree-d polynomial regression model. We can therefore capture all feature interaction terms in a factorization machine with the feature map $h(\boldsymbol{x}) = (h_j(\boldsymbol{x}))_{j\in[\ell]}$ that was introduced in Section 6.4, where each component h_j represents a monomial over $\boldsymbol{X}$.

A factorization machine of rank $\mathcal{R}$ has parameters $\boldsymbol{\theta}$, which consist of a vector of "linear" parameters $\boldsymbol{\theta}_j$ for each $j \in [n]$, as well as a vector of "factorized" parameters $\boldsymbol{\theta}_{j,\ell}$ for each

$j \in [n]$ and $\ell \in [\mathcal{R}]$. Each feature monomial in a factorization machine is multiplied by a polynomial over $\boldsymbol{\theta}$. To capture all parameters uniformly, we introduce a parameter map $g(\boldsymbol{\theta})$, which transforms the model parameters $\boldsymbol{\theta}$ into a ℓ-vector $g(\boldsymbol{\theta}) = (g_j(\boldsymbol{\theta}))_{j\in[\ell]}$, where each g_j is a multivariate polynomial of $\boldsymbol{\theta}$.

A factorization machine of degree d and rank $\mathcal{R}$ is then defined by:

$$\mathsf{FaMa}^r_d(\boldsymbol{x}) = \sum_{i\in[\ell]} \langle g_i(\boldsymbol{\theta}), h_i(\boldsymbol{x})\rangle = \langle g(\boldsymbol{\theta}), h(\boldsymbol{x})\rangle \tag{6.35}$$

Example 6.17. For the interaction term (6.34) between city and color from Example 6.15, the parameter mapping function g would map to one component g_j which encodes the following polynomial over $\boldsymbol{\theta}$:

$$g_j(\boldsymbol{\theta}) = \sum_{r\in[\mathcal{R}]} \boldsymbol{\theta}_{\mathsf{city},r} \otimes \boldsymbol{\theta}_{\mathsf{color},r}.$$

The feature mapping function h would map to the monomial $h_j(\boldsymbol{x}) = \boldsymbol{x}_{\mathsf{city}} \otimes \boldsymbol{x}_{\mathsf{color}}$. □

Learning factorization machines with batch gradient descent

We next show how we can learn factorization machine models with batch gradient descent optimization as shown in Algorithm 6.1. The objective function $J(\boldsymbol{\theta})$ is given by the least squares loss function with ℓ_2 regularization:

$$J(\theta) = \frac{1}{2|D|} \sum_{(\boldsymbol{x},y)\in D} \Big(\sum_{i\in[\ell]} \langle g(\boldsymbol{\theta}), h_i(\boldsymbol{x})\rangle - y\Big)^2 + \frac{\lambda}{2} \|\boldsymbol{\theta}\|_2^2 . \tag{6.36}$$

As for PR_2 models, we again assume, without loss of generality, that the feature map h is extended by an additional term $h_{\ell+1}(\boldsymbol{x}) = y$, and the parameter map is extended by $g_{\ell+1}(\boldsymbol{\theta}) = -1$. We can then simplify and rewrite the objective $J(\boldsymbol{\theta})$ as follows:

$$\begin{aligned}
J(\theta) &= \frac{1}{2|D|} \sum_{(\boldsymbol{x},y)\in D} \Big(\sum_{i\in[\ell+1]} \langle g_i(\boldsymbol{\theta}), h_i(\boldsymbol{x})\rangle \Big)^2 + \frac{\lambda}{2} \|\boldsymbol{\theta}\|_2^2 && (6.37)\\
&= \frac{1}{2|D|} \sum_{(\boldsymbol{x},y)\in D} \Big(\langle g(\boldsymbol{\theta}), h(\boldsymbol{x})\rangle \Big)^2 + \frac{\lambda}{2} \|\boldsymbol{\theta}\|_2^2 && (6.38)\\
&= \frac{1}{2|D|} \sum_{(\boldsymbol{x},y)\in D} g(\boldsymbol{\theta})^\top h(\boldsymbol{x})(\boldsymbol{x})^\top g(\boldsymbol{\theta}) + \frac{\lambda}{2} \|\boldsymbol{\theta}\|_2^2 && (6.39)\\
&= \frac{1}{2|D|} g(\boldsymbol{\theta})^\top \Big(\sum_{(\boldsymbol{x},y)\in D} h(\boldsymbol{x})h(\boldsymbol{x})^\top \Big) g(\boldsymbol{\theta}) + \frac{\lambda}{2} \|\boldsymbol{\theta}\|_2^2 && (6.40)\\
&= \frac{1}{2|D|} g(\boldsymbol{\theta})^\top \boldsymbol{\Sigma} g(\boldsymbol{\theta}) + \frac{\lambda}{2} \|\boldsymbol{\theta}\|_2^2 . && (6.41)
\end{aligned}$$

Note that the matrix $\boldsymbol{\Sigma}$ for factorization machines is identical to the $\boldsymbol{\Sigma}$ for polynomial

regression models, and thus each entry $\boldsymbol{\sigma}_{ij} \in \boldsymbol{\Sigma}$ can be computed with a query of form (6.31). We next show the gradient formulation which can be inferred from (6.41) using chain rule:

$$\boldsymbol{\nabla} J(\theta) = \frac{1}{|D|} \frac{\partial g(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_k}^\top \boldsymbol{\Sigma} g(\boldsymbol{\theta}) + \lambda \boldsymbol{\theta}. \tag{6.42}$$

Equations (6.41) and (6.42) show that the factorization machine can be learned by first computing the $\boldsymbol{\Sigma}$ matrix and then learning the parameters of the models over $\boldsymbol{\Sigma}$, and without scanning over the input data. We next exemplify how we can compute the product between an entry $\boldsymbol{\sigma}_{ij} \in \boldsymbol{\Sigma}$ with the corresponding $g_j(\boldsymbol{\theta})$ without materializing the matrix of parameters defined by the polynomial over $\boldsymbol{\theta}$ that is encoded by g_j.

Example 6.18. Consider the tensor $\boldsymbol{\sigma}_{ij}$, where $i = (\mathsf{city}, \mathsf{country})$ and $j = (\mathsf{city}, \mathsf{store})$ from Example (6.13). The corresponding entry in the parameter map $g_j(\boldsymbol{\theta})$ is defined by:

$$g_j(\boldsymbol{\theta}) = \sum_{r \in [\mathcal{R}]} \boldsymbol{\theta}_{\mathsf{city},r} \otimes \boldsymbol{\theta}_{\mathsf{store},r}. \tag{6.43}$$

The gradient vector (6.42) of the factorization machine requires the computation of the product $\boldsymbol{\sigma}_{ij} g_j(\boldsymbol{\theta})$. This product can be computed as aggregate queries of the form (4.1) and without materializing the matrix defined by $g_j(\boldsymbol{\theta})$ first. We do so by multiplying $\boldsymbol{\sigma}_{ij}$ with each of the terms $\boldsymbol{\theta}_{\mathsf{city},r} \otimes \boldsymbol{\theta}_{\mathsf{store},r}$, one by one for each $r \in [\mathcal{R}]$, and then add up the result. Multiplying the tensor $\boldsymbol{\sigma}_{ij}$ with the first term $\boldsymbol{\theta}_{\mathsf{city},1} \otimes \boldsymbol{\theta}_{\mathsf{store},1}$ corresponds precisely to the following query:

$$\begin{aligned} Q(\mathsf{city}, \mathsf{country}, \mathtt{SUM}(c \cdot p_1 \cdot p_2)) \leftarrow & \ \mathsf{Covar}_{(\mathsf{city},\mathsf{country},\mathsf{store})}(\mathsf{city}, \mathsf{country}, \mathsf{store}), \\ & \boldsymbol{\theta}_{\mathsf{city},1}(\mathsf{city}, p_1), \boldsymbol{\theta}_{\mathsf{store},1}(\mathsf{store}, p_2) \end{aligned}$$

□

Complexity analysis

We next give the runtime complexity for learning factorization machines of degree d and rank $\mathcal{R}$ over feature extraction queries. We assume that the model is learned over the covar matrix $\boldsymbol{\Sigma} = (\boldsymbol{\sigma}_{ij})_{i,j \in [\ell]}$. Let $C_{ij} = C_i \cup C_j$ denote the group-by variables, and $|\boldsymbol{\sigma}_{ij}|$ the number of tuples, in the query of form (6.31) that represents the (i,j)-entry of $\boldsymbol{\Sigma}$.

Theorem 6.19. *Let Q be a feature extraction query over a database instance I where all relations have size at most N. Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ denote the query hypergraph of Q.*

Any factorization machine $\mathsf{FaMa}_d^{\mathcal{R}}(\boldsymbol{x})$ of degree-d and rank-$\mathcal{R}$ can be learned over Q with ξ batch gradient descent iterations in time:

$$O\left(|\mathcal{V}|^2 \cdot |\mathcal{E}| \cdot \sum_{i,j \in [\ell]} (N^{\mathsf{fhtw}_{C_{ij}}(\mathcal{H})} + |\boldsymbol{\sigma}_{ij}|) \cdot \log N + \xi \cdot \mathcal{R} \cdot \sum_{i,j \in [\ell]} |\boldsymbol{\sigma}_{ij}| \right), \tag{6.44}$$

where ℓ is the dimension of the feature mapping vector $h(\boldsymbol{x})$.

Proof. The first part of (6.44) follows directly from Theorem 6.14. The second part corresponds to the computation of the gradient over $\boldsymbol{\Sigma}$, which, for each $r \in [\mathcal{R}]$ can be computed in one pass over each $\boldsymbol{\sigma}_{ij} \in \boldsymbol{\Sigma}$. □

Following Proposition 5.20, our solution for learning a degree-d rank-$\mathcal{R}$ factorization machines over feature extraction query Q takes strictly sublinear time in the size of the output $|Q|$ for infinitely many queries and database instances.

6.6 Model Selection

In this section, we overview model selection, a key challenge in machine learning centered around finding a statistical model from a class of models that best predicts the label. Model selection is a laborious and time-intensive process, since it requires the learning of (potentially many) independent models. We next explain model selection in the context of the models introduced in this chapter.

Consider a feature extraction query Q with n variables that are mapped into ℓ features. The goal of model selection is to find the model over a subset of features that best predicts the label. In addition, the process may also consider models that are learned with different reguarlization terms and/or constants.

In practice, it is often infeasible to compute the model for all possible subsets of features, and thus it is common to use a greedy algorithm instead. A common example is the stepwise model selection algorithm [45], which is an iterative process where each step seeks one feature that is added to or removed from the model until the model accuracy does not improve significantly. There are three flavors of the algorithm:

(1) *Forward selection* starts initially with a model that has no features. At each step, the algorithm then adds the feature to the model that best improves the accuracy of the model under a predefined model fit criterion. Therefore, each step requires the learning of one independent model for each feature under consideration. The process is repeated until the addition of any remaining feature does not significantly improve the model accuracy.

(2) *Backward elimination* initially starts with all features in the model. At each step, the algorithm removes one feature from the model. This feature is chosen based on the accuracy under the predefined model fit criterion, and the process is repeated until no further features can be removed without a statistically significant loss of fit.

(3) *Bidirectional elimination* is a combination of the above, testing at each step for features to be included or excluded.

The model fit criterion to assess the models are typically statistical properties of the model, such as the adjusted R^2 value [45], Akaike information criterion [15], Bayesian information criterion (BIC) [120], or Mallows's Cp [54]. Alternatively, one can also assess the accuracy of the model using a separate test dataset. We next highlight the adjusted R^2 and the Akaike information criterion, the other criteria are addressed similarly.

Adjusted R^2 We first introduce R^2, which is called the coefficient of determination [45]. Let $\mathcal{M}_{\boldsymbol{\theta}}(\boldsymbol{x})$ denote the parameterized model under consideration, and $\mu = \frac{1}{|D|}\sum_{(\boldsymbol{x},y)\in D} y$ be the average over the label in the training dataset. Then the R^2 value is defined as follows:

$$R^2 = 1 - \frac{\mathrm{SS}_{\mathrm{res}}}{\mathrm{SS}_{\mathrm{tot}}} \tag{6.45}$$

$$\mathrm{SS}_{\mathrm{res}} = \sum_{(\boldsymbol{x},y)\in D} (y - \mathcal{M}_{\boldsymbol{\theta}}(\boldsymbol{x}))^2 \tag{6.46}$$

$$\mathrm{SS}_{\mathrm{tot}} = \sum_{(\boldsymbol{x},y)\in D} (y - \mu)^2 \tag{6.47}$$

where $\mathrm{SS}_{\mathrm{res}}$ denotes the *residual sum of squares* and $\mathrm{SS}_{\mathrm{tot}}$ is called the *total sum of squares*.

The R^2 value measures the amount of the variance in the label that can be explained by the model. For instance, suppose $R^2 = 0.52$. Then 52% of the variability of label has been accounted for by the model, and the remaining 48% of the variability is still unaccounted for. The goal is to find the model that accounts for the highest variability in the label, i.e., has the highest R^2 value.

The adjusted R^2 is a modification of the R^2 that adjusts for the number of explanatory terms in a model relative to the number of data points:

$$\bar{R}^2 = 1 - (1 - R^2)\frac{|D| - 1}{|D| - p - 1}, \tag{6.48}$$

where p is the total number of parameters in the model (not including the constant term).

Akaike information criterion The AIC is founded in information theory, and measures the tradeoff between goodness of fit and the number of parameters in the model. The goal is to find the model with the minimum AIC value.

For arbitrary models, the Akaike information criterion (AIC) is defined as follows:

$$\mathrm{AIC} = 2p - 2\ln(\hat{L}), \tag{6.49}$$

where p is the number of parameters in the model and $\hat{L}$ denotes the maximum value of the likelihood function of the model.

For optimization problems with square loss, the AIC can be defined as follows:

$$\text{AIC} = 2p + |D| \ln(\frac{\text{SS}_{\text{res}}}{|D|}) + C, \tag{6.50}$$

where C is a constant independent of the model, and dependent only on the particular data points, i.e., it does not change if the data does not change. Thus, it suffices to compute $\text{AIC} = 2p + |D| \ln(\text{SS}_{\text{res}}/|D|)$ to compare models that are computed over the same data.

6.6.1 Model Selection over Feature Extraction Queries

We consider models with square-loss objective that are learned over databases. For such models, we make two important observations: (1) the data-intensive computation for the model-fit criteria that estimates the accuracy model is provided by the covar matrix that defines the data-intensive computation to learn the model; and (2) we can exploit the fact that aggregation commutes with projection to reuse the aggregates in the covar matrix to compute any model over a subset of the original features.

We explain our first observation in the context of the adjusted R^2 and AIC criteria, for which the data-intensive computation is given by SS_{res} and SS_{tot}. Note that SS_{res} denotes exactly the data-intensive computation to evaluate the objective function (6.3), and thus we can compute it directly over the matrix $\boldsymbol{\Sigma}$. If we learn a (linear or polynomial) regression model than SS_{tot} is defined as follows:

$$\text{SS}_{\text{tot}} = \boldsymbol{\theta}^\top \boldsymbol{\Sigma} \boldsymbol{\theta}$$

For factorization machines, the formulation follows from (6.41).

Similarly, we can compute SS_{tot} using entries in the $\boldsymbol{\Sigma}$ matrix. Consider the following rewriting for SS_{tot},

$$\begin{aligned}
\text{SS}_{\text{tot}} &= \sum_{(\boldsymbol{x},y)\in D} (y-\mu)^2 && (6.51)\\
&= \sum_{(\boldsymbol{x},y)\in D} (y^2 - 2\mu y - \mu^2) && (6.52)\\
&= \Big(\sum_{(\boldsymbol{x},y)\in D} y^2\Big) - 2\mu\Big(\sum_{(\boldsymbol{x},y)\in D} y\Big) - |D|\,\mu^2 && (6.53)\\
&= \Big(\sum_{(\boldsymbol{x},y)\in D} y^2\Big) - \frac{2}{|D|^2}\Big(\sum_{(\boldsymbol{x},y)\in D} y\Big)^2 - \frac{|D|}{|D|^2}\Big(\sum_{(\boldsymbol{x},y)\in D} y\Big)^2 && (6.54)\\
&= \Big(\sum_{(\boldsymbol{x},y)\in D} y^2\Big) - \frac{1}{|D|}\Big(\sum_{(\boldsymbol{x},y)\in D} y\Big)^2. && (6.55)
\end{aligned}$$

This rewriting shows that SS_{tot} can be computed by first computing three aggregates

`SUM(1)`, `SUM(`Y`)`, and `SUM(`$Y \cdot Y$`)` over the training dataset D, all of which are entries in the $\mathbf{\Sigma}$ matrix.

Therefore, for any model that is learned over the covar matrix $\mathbf{\Sigma}$, we can also compute its adjusted R^2 and AIC values over $\mathbf{\Sigma}$ and independent of the input data.

Our second observation is that learning models with different subsets of features is as if the model is learned over a projection of the feature extraction query onto the variables that define those features. Therefore, we can exploit the fact that aggregate queries commute with projection and define the data-intensive computation of the learning problem as a subset of the aggregates that are used to learn the model over all features.

More specifically, this means that we can compute the covar matrix for all features in the model, and then compute models with different subsets of features over the partition of the matrix that defines the features of the model. The remarkable outcome is that we only need to compute the aggregates for the covar matrix once. We can then learn any model over any subset of the features without any additional passes over the input data.

In addition to choosing different subsets of features, we may also learn models with different regularization terms or different labels over the same covar matrix.

6.7 Alternative Optimization Algorithms

In this section, we overview three alternative optimization algorithms which could be used instead of batch gradient descent (BGD) to optimize the objective function (6.3): (1) stochastic gradient descent, (2) coordinate descent, and (3) Quasi-Newton optimization. For each algorithm, we discuss to the extend to which they benefit from a rewriting of the data-intensive computation into aggregate queries.

6.7.1 Stochastic Gradient Descent

A naïve implementation of BGD computes the gradient in a full pass over the training dataset, which can be inefficient in large-scale analytics. In many practical scenarios, it is common to use stochastic gradient descent (SGD) instead, which estimates the gradient with a randomly selected mini-batch of training samples. It is often well-agreed upon that SGD is faster than BGD. The convergence of SGD, however, is noisy, requires careful setting of hyperparameters, and does not achieve the linear asymptotic convergence rate of BGD [26].

A remarkable fact regarding the overall runtime of our approach is that the entire BGD execution can be arbitrarily faster than one SGD epoch over the result of the feature extraction query. The reason is orthogonal to properties of the two gradient descent methods: The complexity of computing the covar matrix needed to compute the gradient can be asymptotically lower than the complexity of computing the result of the feature extraction query. This insight follows from Proposition 5.20, and the fact that the number of iterations for BGD is bounded and independent of the size of the data matrix.

In principle, the mini-batches of SGD could also benefit from factorization. In practice, however, since the mini-batches are supposed to be random, they typically do not have any redundancy that could be exploited via factorization. Therefore, learning a model with SGD over databases would not have any runtime improvements over the conventional approach that first materializes the training dataset and then learns the model.

6.7.2 Coordinate Descent

We next consider coordinate descent optimization, which optimizes each model parameter at a time. This is particularly useful when the one-dimensional problem can be solved analytically. A common example for coordinate descent is to learn lasso regression models, i.e. linear regression models where the objective function is regularized by the ℓ_1-norm [92]. We next show that the data-intensive computation of coordinate descent for learning lasso regression models can be expressed as aggregate queries of the form (4.1).

Recall the objective function for linear regression models from (6.6). For ease of notation, we first consider the case where all variables are continuous. The objective with ℓ_1-regularization is then given by:

$$J(\boldsymbol{\theta}) = \frac{1}{2|D|} \sum_{\boldsymbol{x} \in D} \big(\sum_{j \in [n+1]} \theta_j \cdot x_j \big)^2 + \lambda \left\| \boldsymbol{\theta} \right\|_1 . \tag{6.56}$$

Similarly, the partial derivative with respect to θ_k can be expressed as follows:

$$\frac{\partial}{\partial \theta_k} J(\boldsymbol{\theta}) = \frac{1}{|D|} \sum_{\boldsymbol{x} \in D} \left(\sum_{j \in [n+1]} \theta_j \cdot x_j \right) x_k + \lambda \frac{\partial \left\| \boldsymbol{\theta} \right\|_1}{\partial \theta_k} \tag{6.57}$$

$$= \frac{1}{|D|} \sum_{j \in [n+1]} \theta_j \cdot \sum_{\boldsymbol{x} \in D} (x_j \cdot x_k) + \lambda \frac{\partial \left\| \boldsymbol{\theta} \right\|_1}{\partial \theta_k} \tag{6.58}$$

$$= \frac{1}{|D|} \sum_{j \in [n+1]} \theta_j \cdot \boldsymbol{\sigma}_{jk} + \lambda \frac{\partial \left\| \boldsymbol{\theta} \right\|_1}{\partial \theta_k}. \tag{6.59}$$

Note that the data-intensive computation of (6.59) is exactly the same as for (6.12), where each $\boldsymbol{\sigma}_{jk}$ defines an entry in the $\boldsymbol{\Sigma}$ matrix for linear regression.

The gradient descent algorithm optimizes $J(\boldsymbol{\theta})$ by repeatedly updating all parameters $\boldsymbol{\theta}$ in the direction of the gradient $\boldsymbol{\nabla} J(\boldsymbol{\theta})$. Coordinate descent instead solves for the optimal value for one parameter θ_k, while keeping all other parameters fixed. For lasso regression, we can derive the optimal solution for θ_k analytically, by setting the partial derivative

$\frac{\partial}{\partial \theta_k} J(\boldsymbol{\theta}) = 0$:

$$0 = \frac{1}{|D|} \sum_{j \in [n+1]} \theta_j \cdot \boldsymbol{\sigma}_{jk} + \lambda \frac{\partial \|\boldsymbol{\theta}\|_1}{\partial \theta_k} \tag{6.60}$$

$$= \theta_k \cdot \frac{\boldsymbol{\sigma}_{kk}}{|D|} + \frac{1}{|D|} \sum_{\substack{j \in [n+1] \\ j \neq k}} \theta_j \cdot \boldsymbol{\sigma}_{jk} + \lambda \frac{\partial \|\boldsymbol{\theta}\|_1}{\partial \theta_k} \tag{6.61}$$

Solving (6.61) for θ_k gives the optimal solution for θ_k:

$$\theta_k = S_{\lambda/\boldsymbol{\sigma}_{kk}}(-\frac{1}{\boldsymbol{\sigma}_{kk}} \sum_{\substack{j \in [n+1] \\ j \neq k}} \theta_j \cdot \boldsymbol{\sigma}_{jk}) \tag{6.62}$$

$$= S_{\lambda/\boldsymbol{\sigma}_{kk}}(\theta_k - \frac{1}{\boldsymbol{\sigma}_{kk}} \sum_{j \in [n+1]} \theta_j \cdot \boldsymbol{\sigma}_{jk}) \tag{6.63}$$

where $S_{\lambda/\boldsymbol{\sigma}_{kk}}$ is the soft-thresholding operator:

$$S_\delta(a) = \begin{cases} \text{sign}(a)(|a| - \delta) & \text{if } |a| - \delta > 0 \\ 0 & \text{otherwise} \end{cases} \tag{6.64}$$

The data-intensive computation of the update rule for parameter θ_k is captured by the vector $(\boldsymbol{\sigma}_{jk})_{j \in [n]}$. This is identical to the data-intensive computation of BGD! Therefore, we can solve the optimization problem with coordinate descent by precomputing the $\boldsymbol{\Sigma}$, and then learning the parameters over $\boldsymbol{\Sigma}$ without any additional passes over the input data.

Consider the uniform encoding of categorical and continuous variables of the feature vector $\boldsymbol{x} = (\boldsymbol{x}_i)_{i \in [n]}$ and parameter vector $\boldsymbol{\theta} = (\boldsymbol{\theta}_i)_{i \in [n]}$ from Section 6.3. Using this notation, we can generalize 6.62 to categorical variables as follows:

$$\boldsymbol{\theta}_k^\top = S_{\lambda/\boldsymbol{\sigma}_{kk}}(\boldsymbol{\theta}_k^\top - \boldsymbol{\sigma}_{kk}^{-1} \sum_{j \in [n+1]} \boldsymbol{\theta}_j^\top \cdot \boldsymbol{\sigma}_{jk}), \tag{6.65}$$

where each $\boldsymbol{\sigma}_{jk} = \sum_{(\boldsymbol{x},y)} \boldsymbol{x}_j \boldsymbol{x}_k^\top$ encodes one entry in $\boldsymbol{\Sigma}$, and $S_{\lambda/\boldsymbol{\sigma}_{kk}}$ performs element-wise soft thresholding for each parameter vector $\boldsymbol{\theta}_k$. Note that $\boldsymbol{\sigma}_{kk}$ is a diagonal matrix, so its inverse is trivial to compute.

6.7.3 Quasi-Newton optimization algorithms

All optimization algorithms considered so far are first-order optimization algorithms. Second order optimization algorithms typically have better convergence guarantees, because they take the curvature of the objective function into account. A primary example of a second order optimization algorithm is Newton's method, which like BGD is an iterative optimization algorithm [92].

Let H denote Hessian matrix for $J(\boldsymbol{\theta})$, i.e., the matrix of all second-order derivatives of $J(\boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$. The update rule for Newton's method is defined as follows:

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \alpha \boldsymbol{H}^{-1} \boldsymbol{\nabla} J(\boldsymbol{\theta}). \tag{6.66}$$

The major shortcoming of Newton's method is that computing $\boldsymbol{H}$ and its inverse can be prohibitively expensive. Quasi-Newton methods aim to mitigate this shortcoming by iteratively building an approximation to the Hessian based on the first order derivatives in the gradient vector at each update step. A popular example of Quasi-Newton methods is the L-BFGS algorithm [92], which iteratively updates for each iteration k an estimate of the inverse Hessian $C_k \approx \boldsymbol{H}^{-1}$ as follows:

$$\begin{aligned} \boldsymbol{C}_{k+1} &= (\boldsymbol{I} - \frac{\boldsymbol{s}\boldsymbol{g}^\top}{\boldsymbol{g}^\top \boldsymbol{s}}) \boldsymbol{C}_k (\boldsymbol{I} - \frac{\boldsymbol{g}\boldsymbol{s}^\top}{\boldsymbol{s}^\top \boldsymbol{g}}) + \frac{\boldsymbol{s}\boldsymbol{s}^\top}{\boldsymbol{g}^\top \boldsymbol{s}} \\ \boldsymbol{s} &= \boldsymbol{\theta}^{(k)} - \boldsymbol{\theta}^{(k-1)} \\ \boldsymbol{g} &= \boldsymbol{\nabla} J(\boldsymbol{\theta}^{(k)}) - \boldsymbol{\nabla} J(\boldsymbol{\theta}^{(k-1)}) \end{aligned}$$

where $\boldsymbol{\theta}^{(k)}$ and $\boldsymbol{\nabla}^{(k)} J(\boldsymbol{\theta})$ denotes the parameter vector and respectively gradient at iteration k of the algorithm.

The data-intensive computation of the L-BFGS algorithm is captured by the computation of the gradient $\boldsymbol{\nabla} J(\boldsymbol{\theta})$, which is exactly the same as for BGD. Therefore, we can compute the gradient over the very same $\boldsymbol{\Sigma}$ matrix that we compute for BGD, and then learn the model parameters with L-BFGS optimization independently of the input data.

Chapter 7

Optimization Problems with Non-Polynomial Loss Function

Recall the generic objective function (6.1) for supervised machine learning problems:

$$J(\boldsymbol{\theta}) = \sum_{(\boldsymbol{x},y)\in D} \mathcal{L}\left(\mathcal{M}_{\boldsymbol{\theta}}(\boldsymbol{x}), y\right) + \lambda\Omega(\boldsymbol{\theta}),$$

where $\mathcal{M}_{\boldsymbol{\theta}}(\boldsymbol{x})$ is a model with parameters $\boldsymbol{\theta}$, $\mathcal{L}(a, b)$ is a loss function, Ω is a regularizer, and λ is a constant. We assume that the training dataset D is the result of a feature extraction query Q over database I with relations $R_1, \ldots, R_m$.

In the previous chapter, we considered a class of machine learning models, where the objective is given by the square loss $\mathcal{L}(a, b) = (a - b)^2$. We showed that for learning this class of models over relational databases, the data intensive computation for optimizing the objective $J(\boldsymbol{\theta})$ amounts to computing aggregate queries over the input database.

In many instances, however, the loss function is non-polynomial, either due to the structure of the loss, or the presence of non-polynomial components embedded within the model structure (e.g., ReLU activation function in neural nets) [92].

Examples of commonly used non-polynomial loss functions are: (1) hinge loss, which is used to learn classification models like linear support vector machines (SVM) [92], or for dimensionality reduction with Boolean principal component analysis (PCA) [132]; (2) Huber loss, commonly used to learn regression models that are robust to outliers [92]; (3) scalene loss, which is used to learn quantile regression models [132]; (4) epsilon insensitive loss, which is used to learn SVM regression models [92]; (5) ordinal hinge loss, which is used to learn ordinal regression models or ordinal PCA [132]; and (6) deadzone-linear loss, which is used to compute interval PCA [132]. Boolean, ordinal, and interval PCA belong to the class of Generalized Low Rank Models [132].

The data intensive computation of any optimization problem with the above non-polynomial loss functions can be rewritten into aggregate queries with additive inequalities

of the form (4.1). As a result, for some feature extraction queries we can solve the problem in time sub-linear in $|D|$ and thus faster than the time it takes to materialize the training dataset. We exemplify this reformulation for two popular machine learning problems.

In Section 7.1, we first consider learning linear support vector machines (SVM) for binary classification, where the objective function is defined by the hinge loss function:

$$\mathcal{L}(a, b) = \max(0, 1 - a \cdot b). \tag{7.1}$$

Then, Section 7.2 considers learning robust linear regression models where the loss function is given by Huber loss:

$$\mathcal{L}(a, b) = \begin{cases} \frac{1}{2}(a-b)^2 & \text{if } |a-b| \leq 1, \\ \frac{1}{2}|a-b| - \frac{1}{2} & \text{otherwise.} \end{cases} \tag{7.2}$$

Section 7.3 overviews other non-polynomial loss functions that can benefit from the rewriting of the data-intensive computation. Section 7.4 compare the solutions presented in this chapter with our solution for the square loss problem from Chapter 6. We also present non-polynomial loss functions for which the rewriting into aggregate queries does not bring asymptotic improvements, e.g. logistic loss used for logistic regression.

The results presented in this chapter have previously been published in [5]. The background information is taken from [92], and the survey on generalized low-rank models [132].

7.1 Support Vector Machines

A support vector machine (SVM) classification model is used for binary classification problems where the label $y \in \{\pm 1\}$. In the following, we focus on linear SVM models, but the solution could also be extended to polynomial SVMs. The extension from linear SVMs to polynomial SVMs is analogous to the extension from linear regression to polynomial regression models in Chapter 6.

SVM requires categorical variables to be one-hot encoded. In order to treat continuous and categorical variables uniformly, we use the feature and parameter vector encoding that was introduced in Section 6.1. For a given tuple $(\boldsymbol{x}, y) \in D$, let $\boldsymbol{x} = (\boldsymbol{x}_c)_{c \in [n]}$ denote the feature vector of the model, where each component $\boldsymbol{x}_c$ is also a vector. If X_c is a categorical variable, the vector $\boldsymbol{x}_c = (\mathbf{1}_{X_c = v})_{v \in \mathsf{Dom}(X_c)}$ is the indicator vector derived from the one-hot encoding of X_c. If X_c is a continuous variable, $\boldsymbol{x}_c = (x_c)$ encodes a scalar. Similarly, we define a parameter vector $\boldsymbol{\theta} = (\boldsymbol{\theta}_c)_{c \in [n]}$, where each component $\boldsymbol{\theta}_c$ is a vector of parameters of the same dimension as $\boldsymbol{x}_c$.

The linear SVM model is defined by a linear discriminant function:

$$\mathsf{SVM}(\boldsymbol{x}) = \sum_{j \in [n]} \boldsymbol{\theta}_j^\top \boldsymbol{x}_j = \langle \boldsymbol{\theta}, \boldsymbol{x} \rangle \,, \tag{7.3}$$

where $\langle \boldsymbol{\theta}, \boldsymbol{x} \rangle$ encodes the Frobenius inner product.

The goal is to learn the parameters $\boldsymbol{\theta}$ of the model such that the function $\mathsf{SVM}(\boldsymbol{x})$ separates the data points in D into positive and negative classes with a maximum margin. This can be done by minimizing the objective function (6.1) with hinge loss (7.1). For linear SVM models, the loss function is defined by:

$$\mathcal{L}(y, \mathsf{SVM}(\boldsymbol{x})) = \max(0, 1 - y \cdot \mathsf{SVM}(\boldsymbol{x})). \tag{7.4}$$

The intuition behind hinge loss is as follows: whenever $y = 1$ and $\mathsf{SVM}(\boldsymbol{x}) > 1$ (or equivalently $y = -1$ and $\mathsf{SVM}(\boldsymbol{x}) < -1$) the model correctly classifies the data point $\boldsymbol{x}$ and the hinge loss is zero, i.e. it does not penalize the model. Contrary, if the signs of y and $\mathsf{SVM}(\boldsymbol{x})$ do not match, or if $|\mathsf{SVM}(\boldsymbol{x})| \leq 1$, i.e. the model does not classify with sufficient margin, the hinge loss penalizes the model by $|\mathsf{SVM}(\boldsymbol{x})|$, which is the amount by which the prediction is off.

The objective function for linear SVM classification with ℓ_2-regularization is given by:

$$J(\boldsymbol{\theta}) = \sum_{(\boldsymbol{x}, y) \in D} \max(0, 1 - y(\langle \boldsymbol{\theta}, \boldsymbol{x} \rangle)) + \frac{\lambda}{2} \|\boldsymbol{\theta}\|_2^2 \,. \tag{7.5}$$

The objective with hinge loss is non-differentiable, which means that the gradient $\boldsymbol{\nabla} J(\boldsymbol{\theta})$ is not defined and the optimization problem cannot be solved with standard gradient descent optimization as used for square loss problems in Chapter 6. The objective, however, is convex and admits subgradient vectors. Subgradients generalize the standard notion of gradients and can be used in subgradient-based optimization algorithms.

In the following, we consider the case where the linear SVM is learned with subgradient descent optimization, and show that we can reformulate the data intensive computation of the optimization algorithm as aggregate queries with additive inequalities. We then show that through this reformulation we can learn linear SVM classification models asymptotically faster than the time it takes to materialize the training dataset.

Subgradient-descent optimization for linear SVM classification

Subgradient descent optimization for linear SVM was popularized by the Pegasos [126] algorithm, which showed that subgradient descent optimization can significantly outperform alternative optimization algorithms.

Recall the batch gradient descent algorithm that was presented in Algorithm 6.1. The algorithm can be amended to subgradient descent optimization by updating the parameters

in the direction of the subgradient as opposed to the gradient. For linear SVM classification models, the update rule for each $k \in [n]$ with step size s then becomes:

$$\boldsymbol{\theta}_k = \boldsymbol{\theta}_k - s \cdot \frac{\partial}{\partial \boldsymbol{\theta}_k} J(\boldsymbol{\theta}),$$

where $\frac{\partial}{\partial \boldsymbol{\theta}_k} J(\boldsymbol{\theta})$ defines the partial sub-derivative of the objective function from (7.5):

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_k} = -\sum_{(\boldsymbol{x},y)\in D} y \cdot \boldsymbol{x}_k \cdot \mathbf{1}_{y(\langle \boldsymbol{\theta},\boldsymbol{x}\rangle)\leq 1} + \lambda \boldsymbol{\theta}_k. \tag{7.6}$$

The core of the optimization algorithm is the repeated computation of the objective (7.5) and the partial derivatives (7.6) for each parameter vector $(\boldsymbol{\theta}_j)_{j\in[n]}$. Our observation is that $J(\boldsymbol{\theta})$ and $\frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_j}$ can be reformulated as aggregate queries with additive inequalities of the form (4.1). We first show a rewriting of the objective function (7.5):

$$J(\boldsymbol{\theta}) = \sum_{(\boldsymbol{x},y)\in D} \max\{0, 1 - y(\langle \boldsymbol{\theta},\boldsymbol{x}\rangle)\} + \frac{\lambda}{2}\|\boldsymbol{\theta}\|_2^2 \tag{7.7}$$

$$= \sum_{(\boldsymbol{x},y)\in D} (1 - y(\langle \boldsymbol{\theta},\boldsymbol{x}\rangle) \cdot \mathbf{1}_{y(\langle \boldsymbol{\theta},\boldsymbol{x}\rangle)\leq 1} + \frac{\lambda}{2}\|\boldsymbol{\theta}\|_2^2 \tag{7.8}$$

$$= \sum_{(\boldsymbol{x},y)\in D} \mathbf{1}_{y(\langle \boldsymbol{\theta},\boldsymbol{x}\rangle)\leq 1} - \sum_{(\boldsymbol{x},y)\in D} y \cdot \langle \boldsymbol{\theta},\boldsymbol{x}\rangle \cdot \mathbf{1}_{y(\langle \boldsymbol{\theta},\boldsymbol{x}\rangle)\leq 1} + \frac{\lambda}{2}\|\boldsymbol{\theta}\|_2^2 \tag{7.9}$$

$$= \sum_{\gamma\in\{\pm 1\}} \Bigg(\underbrace{\sum_{(\boldsymbol{x},y)\in D} \mathbf{1}_{y=\gamma} \cdot \mathbf{1}_{\gamma\langle \boldsymbol{\theta},\boldsymbol{x}\rangle\leq 1}}_{\text{Expression } \mathsf{Q}_1} - \gamma \underbrace{\sum_{(\boldsymbol{x},y)\in D} \langle \boldsymbol{\theta},\boldsymbol{x}\rangle \cdot \mathbf{1}_{y=\gamma} \cdot \mathbf{1}_{\gamma\langle \boldsymbol{\theta},\boldsymbol{x}\rangle\leq 1}}_{\text{Expression } \mathsf{Q}_2} \Bigg) + \frac{\lambda}{2}\|\boldsymbol{\theta}\|_2^2 \tag{7.10}$$

Expressions Q_1 and Q_2 represent aggregate queries with inequalities. To express them in the query language from Section 4.2, we recall the functional encoding of the products $\boldsymbol{\theta}_j^\top \boldsymbol{x}_j$ that was introduced for linear regression models in Section 6.3. Consider a tuple $(\boldsymbol{x}, y) \in D$. For each variable X_j, we use function $f_j(x_j)$ to encode the product $\boldsymbol{\theta}_j^\top \boldsymbol{x}_j$. For continuous feature X_j, the function $f_j(x_j) = \theta_{X_j} \cdot x_j$ returns the product of the scalar parameter for X_j and the value for X_j. For categorical variable X_j, the function $f_j(x_j) = \theta_{X_j}(x_j)$ returns the parameter for the value $x_j \in \mathsf{Dom}(X_j)$. Under this encoding, we can now define the queries that compute expressions Q_1 and Q_2.

For each $\gamma \in \{\pm 1\}$, expression Q_1 is represented with the following `COUNT`:

$$\mathsf{Q}_1(\texttt{SUM(1)}) = R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}), \gamma \sum_{j\in[n]} f_j(X_j) \leq 1, Y = \gamma. \tag{7.11}$$

Similarly, Q_2 can be expressed with the following SUM-query:

$$\mathsf{Q}_2(\texttt{SUM}(\sum_{j\in[n]} f_j(X_j))) = R_1(\omega_{R_1}),\ldots,R_m(\omega_{R_m}),\gamma \sum_{j\in[n]} f_j(X_j) \leq 1, Y = \gamma. \tag{7.12}$$

Queries (7.11) and (7.12) are both of the from (4.1) and have one additive inequality over all variables in $\mathcal{V}$. We next show a similar rewriting for the partial subderivative $\frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_j}$:

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_j} = -\sum_{(\boldsymbol{x},y)\in D} y\cdot \boldsymbol{x}_j \cdot \mathbf{1}_{y(\langle \boldsymbol{\theta},\boldsymbol{x}\rangle)\leq 1} + \lambda\theta_j \tag{7.13}$$

$$= -\sum_{\gamma\in\{\pm 1\}} \gamma \underbrace{\sum_{(\boldsymbol{x},y)\in D} \boldsymbol{x}_j \cdot \mathbf{1}_{\gamma(\langle \boldsymbol{\theta},\boldsymbol{x}\rangle)\leq 1}}_{\text{Expression } \mathsf{Q}_3} + \lambda\theta_j. \tag{7.14}$$

If X_j is a continuous variable, then expression Q_3 can be computed as an aggregate query with one additive inequality that is of the form (4.1):

$$\mathsf{Q}_3(\texttt{SUM}(\sum_{k\in[n]} f_k(X_k)\cdot X_j)) = R_1(\omega_{R_1}),,\ldots,R_m(\omega_{R_m}),\gamma \sum_{j\in[n]} f_j(X_j) \leq \gamma, Y = \gamma. \tag{7.15}$$

If X_j is a categorical variable, the expression computes a vector. In this case, the expression can be computed as an aggregate query of the form (4.1) which keeps X_j as a group-by variable:

$$\mathsf{Q}_3(X_j,\texttt{SUM}(\sum_{k\in[n]} f_k(X_k))) = R_1(\omega_{R_1}),\ldots,R_m(\omega_{R_m}),\gamma \sum_{j\in[n]} f_j(X_j) \leq \gamma, Y = \gamma. \tag{7.16}$$

Complexity analysis

Theorem 7.1. *Let Q be a feature extraction query with n features that is computed over input database I, where all relations have size at most N. Let $\mathcal{H} = (\mathcal{V}, \mathcal{E} = \mathcal{E}_R \cup \mathcal{E}_\infty)$ denote the query hypergraph of Q.*

For any linear SVM classification model $\mathsf{SVM}(\boldsymbol{x})$ that is learned over Q, we can compute each batch subgradient descent update step in time:

$$O(|\mathcal{V}|^2 \cdot |\mathcal{E}| \cdot n \cdot N^{\mathsf{r\text{-}fhtw}(\mathcal{H})} \cdot \log N).$$

Proof. The above rewritings show that both $J(\boldsymbol{\theta})$ and $\boldsymbol{\nabla} J(\boldsymbol{\theta})$ can be rewritten into aggregate queries of the form (4.1), which have one additive inequality $|\mathcal{E}_\infty| = 1$ (The condition $y = \gamma$ can be absorbed by one relation that contains variable Y). In addition, the queries have at most one free variable, which implies two things: (1) any tree decomposition for the query is trivially a free-connex tree decomposition, and (2) following Proposition 5.20 their output size is bounded by N.

The overall runtime bound to learn linear SVM classification models follows from Theorem 5.18, which states that each query φ of the form (4.1) with on free variable can be computed in time $O(|\mathcal{V}|^2 \cdot |\mathcal{E}| \cdot s \cdot (N^{\mathsf{r\text{-}fhtw}(\mathcal{H})} + |\varphi|) \cdot \log N)$, where s is number of products in the user defined aggregate α. For the queries that define $J(\boldsymbol{\theta})$ and $\boldsymbol{\nabla} J(\boldsymbol{\theta})$, $s = O(n)$. □

The number of iterations in the subgradient descent algorithm is independent of $|D|$. Therefore, following Proposition 5.20, we can compute the linear SVM classification model in time sub-linear to the time it takes to materialize the training dataset D.

7.2 Robust Linear Regression

Recall the linear regression model that was introduced in Section 6.3 which estimates the continuous label $y \in \mathbb{R}$:

$$\mathsf{LR}(\boldsymbol{x}) = \sum_{j \in [n]} \boldsymbol{\theta}_j^\top \boldsymbol{x} = \langle \boldsymbol{\theta}, \boldsymbol{x} \rangle \,. \tag{7.17}$$

Section 6.3 presents a solution to learn LR models with square loss. In this section, we consider an alternative approach which learns the model with the Huber loss function (7.2). For linear regression models, Huber loss is given by:

$$\mathcal{L}(\mathsf{LR}(\boldsymbol{x}), y) = \begin{cases} \frac{1}{2}(\mathsf{LR}(\boldsymbol{x}) - y)^2 & \text{if } |\mathsf{LR}(\boldsymbol{x}) - y| \leq 1, \\ \frac{1}{2}|\mathsf{LR}(\boldsymbol{x}) - y| - \frac{1}{2} & \text{otherwise.} \end{cases} \tag{7.18}$$

Huber loss is equivalent to the square loss when $|\mathsf{LR}(\boldsymbol{x}) - y| \leq 1$ and to the absolute loss otherwise.[1] There are two advantages for using Huber loss for linear regression: (1) in contrast to the absolute loss, Huber loss is differentiable at all points, and (2) in comparison to square loss, the absolute loss assigns less weight to points that are far from the model prediction, which has the effect that a learned model tends to be more robust to outliers in the dataset (hence the name robust linear regression).

The objective function $J(\boldsymbol{\theta})$ with ℓ_2-regularization for robust linear regression models is defined as follows:

$$J(\boldsymbol{\theta}) = \frac{\lambda}{2} \|\boldsymbol{\theta}\|_2^2 + \frac{1}{2|D|} \sum_{(\boldsymbol{x}, y) \in D} \begin{cases} (\mathsf{LR}(\boldsymbol{x}) - y)^2 & \text{if } |\mathsf{LR}(\boldsymbol{x}) - y| \leq 1, \\ |\mathsf{LR}(\boldsymbol{x}) - y| - 1 & \text{otherwise.} \end{cases} \tag{7.19}$$

Since the objective function is convex and differentiable, we learn the model parameters with the batch gradient descent optimization algorithm presented in Algorithm 6.1. Recall that batch gradient descent repeatedly updates each parameter $(\boldsymbol{\theta}_k)_{k \in [n]}$ in the direction

[1] Without loss of generality, we use a simplified Huber loss. In general, the threshold between absolute and square loss is given by a constant δ and the absolute loss is $\frac{\delta}{2}|a - b| - \frac{\delta^2}{2}$.

of the partial derivative $\frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_k}$ until convergence. In this section, we focus on the core computation of the algorithm, which is the repeated computation of $J(\boldsymbol{\theta})$ and $\frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_k}$. We show that their data dependent computation can be expressed as aggregate queries with inequalities.

As in Section 6.3, we extend, without loss of generality, the feature vector $\boldsymbol{x}$ with an additional term $\boldsymbol{x}_{n+1} = y$ and define corresponding parameter $\boldsymbol{\theta}_{n+1} = -1$. Then we can simplify the expression for $J(\boldsymbol{\theta})$ as follows:

$$J(\boldsymbol{\theta}) = \frac{\lambda}{2}\left\|\boldsymbol{\theta}\right\|_2^2 + \frac{1}{2|D|}\sum_{(\boldsymbol{x},y)\in D}\Big(\sum_{i\in[n+1]}\boldsymbol{\theta}_i^\top \boldsymbol{x}_i\Big)^2 \cdot \mathbf{1}_{\left|\sum_{i\in[n+1]}\boldsymbol{\theta}_i^\top \boldsymbol{x}_i\right|\leq 1} + \frac{1}{2|D|}\sum_{(\boldsymbol{x},y)\in D}\Big(\Big|\sum_{i\in[n+1]}\boldsymbol{\theta}_i^\top \boldsymbol{x}_i\Big| - 1\Big) \cdot \mathbf{1}_{\left|\sum_{i\in[n+1]}\boldsymbol{\theta}_i^\top \boldsymbol{x}_i\right|>1}. \tag{7.20}$$

Each of the two cases in (7.20) can be expressed as aggregate queries with additive inequalities of the form (4.1). We first show the rewriting for the square loss part of the objective:

$$\sum_{(\boldsymbol{x},y)\in D}\Big(\sum_{i\in[n+1]}\boldsymbol{\theta}_i^\top \boldsymbol{x}_i\Big)^2 \cdot \mathbf{1}_{\left|\sum_{i\in[n+1]}\boldsymbol{\theta}_i^\top \boldsymbol{x}_i\right|\leq 1} \tag{7.21}$$

$$= \sum_{(\boldsymbol{x},y)\in D}\Big(\sum_{\substack{(i,j)\in[n+1]\\ i\leq j}}\boldsymbol{\theta}_i^\top \boldsymbol{x}_i\boldsymbol{x}_j^\top \boldsymbol{\theta}_j\Big) \cdot \mathbf{1}_{\left|\sum_{i\in[n+1]}\boldsymbol{\theta}_i^\top \boldsymbol{x}_i\right|\leq 1} \tag{7.22}$$

$$= \sum_{(\boldsymbol{x},y)\in D}\Big(\sum_{\substack{(i,j)\in[n+1]\\ i\leq j}}\boldsymbol{\theta}_i^\top \boldsymbol{x}_i\boldsymbol{x}_j^\top \boldsymbol{\theta}_j\Big) \cdot \mathbf{1}_{\sum_{i\in[n+1]}\boldsymbol{\theta}_i^\top \boldsymbol{x}_i<0} \cdot \mathbf{1}_{\sum_{i\in[n+1]}\boldsymbol{\theta}_i^\top \boldsymbol{x}_i\geq -1} + \sum_{(\boldsymbol{x},y)\in D}\Big(\sum_{\substack{(i,j)\in[n+1]\\ i\leq j}}\boldsymbol{\theta}_i^\top \boldsymbol{x}_i\boldsymbol{x}_j^\top \boldsymbol{\theta}_j\Big) \cdot \mathbf{1}_{\sum_{i\in[n+1]}\boldsymbol{\theta}_i^\top \boldsymbol{x}_i\leq 1} \cdot \mathbf{1}_{\sum_{i\in[n+1]}\boldsymbol{\theta}_i^\top \boldsymbol{x}_i\geq 0}. \tag{7.23}$$

Recall the functional encoding for the products $\boldsymbol{\theta}_i^\top \boldsymbol{x}_i$ that was shown for linear SVM above. Under this encoding, we can rewrite the inner part of expression (7.23) as follows:

$$\sum_{\substack{(i,j)\in[n+1]\\ i\leq j}}\boldsymbol{\theta}_i^\top \boldsymbol{x}_i\boldsymbol{x}_j^\top \boldsymbol{\theta}_j = \sum_{\substack{(i,j)\in[n+1]\\ i\leq j}} f_j(\boldsymbol{x}_j)\cdot f_i(\boldsymbol{x}_i). \tag{7.24}$$

The rewriting is sum over products of two unary functions, which is exactly the type of aggregate that is supported by the class of aggregate queries with inequalities for Section 5.4. We next show a rewriting of the first part of (7.23) as an aggregate query with two additive

inequalities, the second part can be rewritten similarly:

$$Q_1(\texttt{SUM}(\alpha)) = R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}), \sum_{i,\in[n+1]} f_i(x_i) \leq 0, \sum_{i,\in[n+1]} f_i(x_i) \geq -1, \tag{7.25}$$

$$\text{where } \alpha = \sum_{\substack{(i,j)\in[n+1]\\ i\leq j}} f_j(x_j) \cdot f_i(x_i).$$

We next show a similar rewriting for the absolute part of the objective (7.20):

$$\begin{aligned}
&\sum_{(\boldsymbol{x},y)\in D} \Big(\Big| \sum_{i\in[n+1]} \boldsymbol{\theta}_i^\top \boldsymbol{x}_i \Big| - 1 \Big) \cdot \mathbf{1}_{\left|\sum_{i\in[n+1]} \boldsymbol{\theta}_i^\top \boldsymbol{x}_i\right|>1} \\
&= \sum_{(\boldsymbol{x},y)\in D} \Big(\sum_{i\in[n+1]} \boldsymbol{\theta}_i^\top \boldsymbol{x}_i - 1 \Big) \cdot \mathbf{1}_{\sum_{i\in[n+1]} \boldsymbol{\theta}_i^\top \boldsymbol{x}_i>1} - \sum_{(\boldsymbol{x},y)\in D} \Big(\sum_{i\in[n+1]} \boldsymbol{\theta}_i^\top \boldsymbol{x}_i + 1 \Big) \cdot \mathbf{1}_{\sum_{i\in[n+1]} \boldsymbol{\theta}_i^\top \boldsymbol{x}_i<1} \\
&= \sum_{(\boldsymbol{x},y)\in D} \Big(\sum_{i\in[n+1]} \boldsymbol{\theta}_i^\top \boldsymbol{x}_i \Big) \cdot \mathbf{1}_{\sum_{i\in[n+1]} \boldsymbol{\theta}_i^\top \boldsymbol{x}_i>1} - \sum_{(\boldsymbol{x},y)\in D} \Big(\sum_{i\in[n+1]} \boldsymbol{\theta}_i^\top \boldsymbol{x}_i \Big) \cdot \mathbf{1}_{\sum_{i\in[n+1]} \boldsymbol{\theta}_i^\top \boldsymbol{x}_i<1} - \\
&\quad \sum_{(\boldsymbol{x},y)\in D} \mathbf{1}_{\sum_{i\in[n+1]} \boldsymbol{\theta}_i^\top \boldsymbol{x}_i<1} - \sum_{(\boldsymbol{x},y)\in D} \mathbf{1}_{\sum_{i\in[n+1]} \boldsymbol{\theta}_i^\top \boldsymbol{x}_i>1}.
\end{aligned} \tag{7.26}$$

Each of these four terms encode aggregate queries with additive inequalities of the form (4.1). We show the queries for the first two terms of the expression, the other two terms have the same body and compute a simple `COUNT` aggregate:

$$Q_2(\texttt{SUM}(\textstyle\sum_{i\in[n]} f_i(x_i))) = R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}), \sum_{i,\in[n+1]} f_i(x_i) > 1$$

$$Q_3(\texttt{SUM}(\textstyle\sum_{i\in[n]} f_i(x_i))) = R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}), \sum_{i,\in[n+1]} f_i(x_i) < 1$$

We next show a similar rewriting for the partial derivative of the objective with respect to parameter $\boldsymbol{\theta}_j$:

$$\begin{aligned}
\frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_j} &= \lambda \boldsymbol{\theta}_j + \frac{1}{|D|} \sum_{(\boldsymbol{x},y)\in D} \Big(\sum_{i\in[n+1]} \boldsymbol{\theta}_i^\top \boldsymbol{x}_i \Big) \boldsymbol{x}_j^\top \cdot \mathbf{1}_{\left|\sum_{i\in[n+1]} \boldsymbol{\theta}_i^\top \boldsymbol{x}_i\right|\leq 1} + \qquad (7.27) \\
&\qquad \frac{1}{2} \Big(\boldsymbol{x}_j \cdot \mathbf{1}_{\sum_{i\in[n+1]} \boldsymbol{\theta}_i^\top \boldsymbol{x}_i>0} - \boldsymbol{x}_j \cdot \mathbf{1}_{\sum_{i\in[n+1]} \boldsymbol{\theta}_i^\top \boldsymbol{x}_i<0} \Big) \cdot \mathbf{1}_{\left|\sum_{i\in[n+1]} \boldsymbol{\theta}_i^\top \boldsymbol{x}_i\right|>1} \\
&= \lambda \boldsymbol{\theta}_j + \frac{1}{|D|} \sum_{(\boldsymbol{x},y)\in D} \Big(\sum_{i\in[n+1]} \boldsymbol{\theta}_i^\top \boldsymbol{x}_i \Big) \boldsymbol{x}_j^\top \cdot \mathbf{1}_{\left|\sum_{i\in[n+1]} \boldsymbol{\theta}_i^\top \boldsymbol{x}_i\right|\leq 1} + \qquad (7.28) \\
&\qquad \frac{1}{2|D|} \sum_{(\boldsymbol{x},y)\in D} \boldsymbol{x}_j \cdot \mathbf{1}_{\sum_{i\in[n+1]} \boldsymbol{\theta}_i^\top \boldsymbol{x}_i>1} - \frac{1}{2|D|} \sum_{(\boldsymbol{x},y)\in D} \boldsymbol{x}_j \cdot \mathbf{1}_{\sum_{i\in[n+1]} \boldsymbol{\theta}_i^\top \boldsymbol{x}_i<-1}.
\end{aligned}$$

The first term of (7.28) can be rewritten into two expressions similar to (7.23). The partial derivate of $J(\boldsymbol{\theta})$ thus consists of four terms, where each term can be expressed as an aggregate query with inequalities. We show this rewriting for the last term in the expression,

the other terms can be rewritten similarly.

If X_j is a continuous variable, the last term of (7.28) (without the division by $\frac{1}{2|D|}$) computes a scalar, which can be represented with the following query:

$$Q_4(\texttt{SUM}(X_j)) = R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}), \sum_{i,\in[n+1]} f_i(x_i) < -1.$$

If X_j is a categorical variable, the last term of (7.28) (without the division by $\frac{1}{2|D|}$) computes a vector, which can be represented with the following query:

$$Q_5(X_j, \texttt{SUM}(1)) = R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}), \sum_{i,\in[n+1]} f_i(x_i) < -1.$$

There is a notable difference between the queries used to learn robust linear regression models, and the queries that are computed to learn linear regression models with square-loss as shown in Section 6.3: The additive inequalities in the queries for robust linear regression models depend on the parameters of the model. As a result, the queries evaluation cannot be decoupled from the optimization algorithm used to learn the parameters. This means that we need to recompute the queries for each gradient-descent optimization step.

Complexity Analysis

Theorem 7.2. *Let Q be a feature extraction query with n features that is computed over input database I, where all relations have size at most N. Let $\mathcal{H} = (\mathcal{V}, \mathcal{E} = \mathcal{E}_R \cup \mathcal{E}_\infty)$ denote the query hypergraph of Q.*

For any robust linear regression model $LR(\boldsymbol{x})$ that is learned over Q, we can compute each batch gradient descent iteration in time:

$$O(|\mathcal{V}|^2 \cdot |\mathcal{E}| \cdot n^2 \cdot N^{\mathsf{r\text{-}fhtw}(\mathcal{H})} \cdot \log N). \tag{7.29}$$

Proof. Batch gradient descent optimization requires for each iteration $\ell \in [\xi]$ the computation of the objective $J(\boldsymbol{\theta})$ and gradient $\boldsymbol{\nabla} J(\boldsymbol{\theta})$.

We have shown that we can rewrite of the objective $J(\boldsymbol{\theta})$ and gradient $\boldsymbol{\nabla} J(\boldsymbol{\theta})$ into n aggregate queries of the form (4.1) with at most $|\mathcal{E}_\infty| = 2$ inequality hyperedges (i.e. $k = 1$). In addition, each query can have at most one free variable, which implies (1) that the set of all tree decompositions of Q is trivially free-connex, and (2) that output size is bounded by N (see Proposition 5.20).

The overall runtime bound for computing $J(\boldsymbol{\theta})$ and $\boldsymbol{\nabla} J(\boldsymbol{\theta})$ thus directly follows from Theorem 5.18 which states that each query φ of the form (4.1) with one free variable can be computed in time $O(|\mathcal{V}|^2 \cdot |\mathcal{E}| \cdot s \cdot (N^{\mathsf{r\text{-}fhtw}(\mathcal{H})} \cdot |\varphi|) \cdot \log^k N)$, where s is the number of products in the user defined aggregate α. For the queries that define $J(\boldsymbol{\theta})$ and $\boldsymbol{\nabla} J(\boldsymbol{\theta})$, $s = O(n^2)$, because the aggregate expression for query (7.25) is a summation over $O(n^2)$ products of

functions. □

The number of iterations in the batch gradient descent optimization algorithm is independent of $|D|$. Therefore, following Proposition 5.20, we can compute the robust linear regression model asymptotically faster than the time it takes to materialize the feature extraction query that defines training dataset D.

7.3 Other Non-Polynomial Loss Functions

In this section, we overview the following non-polynomial loss functions: (1) epsilon insensitive loss; (2) ordinal hinge loss; and (3) scalene loss. For each function, we define the loss function $\mathcal{L}$, the corresponding objective function $J(\boldsymbol{\theta})$, and the partial (sub)derivative $\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_j}$ which is used in (sub)gradient-based optimization algorithms. In the derivations for the objective $J(\boldsymbol{\theta})$, we focus on the loss function and ignore the regularizer for better readability.

As in the previous sections, the objective and (sub)dervative can be reformulated into a few queries of the form (4.1). Instead of writing out the expressions explicitly, we annotate those terms that can be reformulated. The actual reformulation should be clear from the examples in Sections 7.1 and 7.2.

Epsilon insensitive loss

The epsilon insensitive loss function [92] is defined as:

$$\mathcal{L}(a,b) = \begin{cases} 0 & \text{if } |a-b| \leq \epsilon, \\ |a-b| - \epsilon & \text{otherwise.} \end{cases}$$

This loss function is used to learn SVM regression models. We consider learning a linear regression model $\mathsf{LR}(\boldsymbol{x}) = \langle \boldsymbol{\theta}, \boldsymbol{x} \rangle$. The objective function and the corresponding partial subderivative with respect to θ_j are given by:

$$\begin{aligned} J(\boldsymbol{\theta}) &= \sum_{(\boldsymbol{x},y)\in D} (|y - \langle \boldsymbol{\theta}, \boldsymbol{x} \rangle| - \epsilon) \cdot \mathbf{1}_{|y-\langle \boldsymbol{\theta},\boldsymbol{x}\rangle|>\epsilon} \\ &= \underbrace{\sum_{(\boldsymbol{x},y)\in D} (y - \langle \boldsymbol{\theta}, \boldsymbol{x} \rangle - \epsilon) \cdot \mathbf{1}_{y-\langle \boldsymbol{\theta},\boldsymbol{x}\rangle|>\epsilon}}_{\text{Query of the form (4.1)}} + \underbrace{\sum_{(\boldsymbol{x},y)\in D} (\langle \boldsymbol{\theta}, \boldsymbol{x} \rangle - y - \epsilon) \cdot \mathbf{1}_{\langle \boldsymbol{\theta},\boldsymbol{x}\rangle-y>\epsilon}}_{\text{Query of the form (4.1)}} \\ \frac{\partial J(\boldsymbol{\theta})}{\partial \theta_j} &= \underbrace{\sum_{(\boldsymbol{x},y)\in D} x_j \cdot \mathbf{1}_{\langle \boldsymbol{\theta},\boldsymbol{x}\rangle-y<\epsilon}}_{\text{Query of form (4.1)}} - \underbrace{\sum_{(\boldsymbol{x},y)\in D} x_j \cdot \mathbf{1}_{y-\langle \boldsymbol{\theta},\boldsymbol{x}\rangle>\epsilon}}_{\text{Query of form (4.1)}} \end{aligned}$$

Ordinal hinge loss

The ordinal hinge loss [132] is defined as:

$$\begin{aligned}\mathcal{L}(a,b) &= \sum_{t=1}^{a-1} \max(0, 1-b+t) + \sum_{t=a+1}^{d} \max(0, 1+b-t) \\ &= \sum_{t=1}^{d} \max(0, 1-b+t) \cdot \mathbf{1}_{t<a} + \max(0, 1+b-t) \cdot \mathbf{1}_{t>a} \\ &= \sum_{t=1}^{d} (1-b+t) \cdot \mathbf{1}_{t<a} \cdot \mathbf{1}_{b<t+1} + (1+b-t) \cdot \mathbf{1}_{t>a} \cdot \mathbf{1}_{b>1-t}.\end{aligned}$$

The loss function is used to learn ordinal regression models or ordinal PCA [132]. A linear ordinal regression model is a linear function $\mathsf{LR}(\boldsymbol{x}) = \langle \boldsymbol{\theta}, \boldsymbol{x} \rangle$ which predicts an ordinal label $y \in [d]$. The objective function and the partial subderivative are given by:

$$J(\boldsymbol{\theta}) = \sum_{t \in [d]} \underbrace{\sum_{(\boldsymbol{x},y) \in D} (1 - \langle \boldsymbol{\theta}, \boldsymbol{x} \rangle + t) \cdot \mathbf{1}_{\langle \boldsymbol{\theta}, \boldsymbol{x} \rangle < 1+t} \cdot \mathbf{1}_{y<t}}_{O(n) \text{ Queries of form (4.1)}} + \underbrace{\sum_{(\boldsymbol{x},y) \in D} (1 + \langle \boldsymbol{\theta}, \boldsymbol{x} \rangle - t) \cdot \mathbf{1}_{\langle \boldsymbol{\theta}, \boldsymbol{x} \rangle > t-1} \cdot \mathbf{1}_{y>t}}_{O(n) \text{ Queries of form (4.1)}}$$

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_j} = \sum_{t \in [d]} \underbrace{\sum_{(\boldsymbol{x},y) \in D} x_j \cdot \mathbf{1}_{\langle \boldsymbol{\theta}, \boldsymbol{x} \rangle > t-1} \cdot \mathbf{1}_{y>t}}_{\text{Query of form (4.1)}} - \underbrace{\sum_{(\boldsymbol{x},y) \in D} x_j \cdot \mathbf{1}_{\langle \boldsymbol{\theta}, \boldsymbol{x} \rangle < 1+t} \cdot \mathbf{1}_{y<t}}_{\text{Query of form (4.1)}}$$

Scalene loss

The scalene loss function [132] is defined as:

$$\begin{aligned}\mathcal{L}(a,b) &= \alpha \cdot \max(0, a-b) + (1-\alpha) \cdot \max(0, b-a) \\ &= \alpha \cdot (a-b) \cdot \mathbf{1}_{a>b} + (1-\alpha) \cdot (b-a) \cdot \mathbf{1}_{b>a},\end{aligned}$$

where $\alpha \in (0,1)$ is a constant.

The loss function is used to learn quantile regression models. We again consider a linear regression model $\mathsf{LR}(\boldsymbol{x}) = \langle \boldsymbol{\theta}, \boldsymbol{x} \rangle$. The objective function and the partial subderivative with

respect to θ_j are given by:

$$J(\boldsymbol{\theta}) = \alpha \underbrace{\sum_{(\boldsymbol{x},y)\in D} (y - \langle \boldsymbol{\theta}, \boldsymbol{x} \rangle) \cdot \mathbf{1}_{y > \langle \boldsymbol{\theta}, \boldsymbol{x} \rangle}}_{\text{Query of form (4.1)}} + (1-\alpha) \underbrace{\sum_{(\boldsymbol{x},y)\in D} (\langle \boldsymbol{\theta}, \boldsymbol{x} \rangle - y) \cdot \mathbf{1}_{\langle \boldsymbol{\theta}, \boldsymbol{x} \rangle > xy}}_{\text{Query of form (4.1)}}$$

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_j} = (1-\alpha) \underbrace{\sum_{(\boldsymbol{x},y)\in D} x_j \cdot \mathbf{1}_{\langle \boldsymbol{\theta}, \boldsymbol{x} \rangle > y}}_{\text{Query of form (4.1)}} - \alpha \underbrace{\sum_{(\boldsymbol{x},y)\in D} x_j \cdot \mathbf{1}_{y > \langle \boldsymbol{\theta}, \boldsymbol{x} \rangle}}_{\text{Query of form (4.1)}}$$

7.4 Discussion

In this section, we compare our solution for optimization problem with non-polynomial loss functions with the solution for square-loss functions from Chapter 6. We also show that there are non-polynomial loss functions for which the rewriting of the data-intensive computation into aggregate queries does not achieve asymptotic runtime improvements.

Comparison of solutions for square-loss and non-polynomial loss

The theoretical analysis of our structure-aware solutions for optimization problems with square loss and with non-polynomial loss are comparable. The solutions achieve asymptotic improvements over the mainstream approach that first materializes the feature extraction query and then learns the model on the result. Yet from a practical perspective, there are stark differences between the two approaches. In particular, there are two reasons why our solution for square loss is superior in practice to the solutions for non-polynomial loss.

The first reason is that our solution for non-polynomial loss functions does not decouple the model parameters from data-intensive computation. Therefore, the aggregate queries that define the data intensive computation need to be recomputed for each (sub)gradient optimization step. In contrast, for optimization problems with square loss, we can capture the data-intensive computation in form of the covar matrix, which can be computed once and reused for all gradient descent optimization steps.

Second, the queries that define the data-intensive computation for the problems with non-polynomial loss functions have additive inequalities that involve all variables in the feature extraction query. Therefore, even under relaxed tree decompositions, the tree decompositions used for the evaluation of the queries that define the data intensive computation can have at most two bags. Any feature extraction query over databases with more than two relations would thus have to first compute the two bags, before evaluating the query result. The bags are computed by joining the input relations. The question whether this evaluation strategy can achieve significant performance improvements in practice over the naïve evaluation of the queries over the join result remains open.

Logistic Loss

There are non-polynomial loss functions for which the rewriting of the data-intensive computation into aggregate queries does not achieve asymptotic runtime improvements. This includes the class of exponential functions used to learn generalized linear models [94]. One popular loss function in this class is the logistic loss function which is used to learn logistic regression models [92].

The logistic loss function requires the computation of an exponential n-ary function:

$$f_{\boldsymbol{\theta}}(\boldsymbol{x}) = \frac{e^{\langle \boldsymbol{\theta}, \boldsymbol{x} \rangle}}{1 + e^{\langle \boldsymbol{\theta}, \boldsymbol{x} \rangle}} \tag{7.30}$$

This function cannot be factorized by exploiting distributivity of products over summations, and therefore we cannot push the computation of this function past joins. For learning logistic regression models over arbitrary feature extraction queries, we therefore need to compute the query result first and then learn the model.

In many practical scenarios, however, feature extraction queries have a star schema with a large fact table in the middle and several dimension tables (see the datasets in Chapter 18). For such feature extraction queries, it is possible to factorize the computation of the inner function $\langle \boldsymbol{\theta}, \boldsymbol{x} \rangle$ by pre-aggregating the inner product over the dimension tables. The logistic regression model can then be learned by having one pass over the fact table, which looks up the aggregates in the dimension tables [79]. As a result, we can effectively share computation over the smaller dimension tables, but the approach does not achieve asymptotic runtime improvements.

Alternatively, it is also possible to exploit geometric and statistical properties of the learning problem to approximate the solution. For instance, we can replace the logistic loss function with the piecewise sigmoid function, which could then be learned with the techniques presented in this chapter. Another option is to the learn the model over small coreset of the training data. In Section 9.2.2, we present a novel clustering algorithm that approximates the k-means solution by clustering over a small coreset instead of the full result of the feature extraction query. The coreset guarantees that the final result is a constant factor approximation of the k-means objective, and can be computed as a batch of aggregate queries (without additive inequalities). Thus, we are able to compute a good approximation to the objective, while benefiting from the asymptotic runtime improvements brought by factorized query evaluation. Such techniques could also be applied to other learning problems, including the learning of logistic regression problems.

Chapter 8

Decision Trees

Decision trees are popular machine learning models that use trees with inner nodes representing conditional control statements to model decisions and their consequences. Leaf nodes represent predictions for the label. Decision trees have several desirable properties: (1) decision tree learners automatically select the most relevant features, and do not require extensive model selection; (2) they do not require extensive hyperparameter tuning; (3) they do not require an encoding of categorical attributes; and (4) they are interpretable. It is due to these properties that decision trees are used in many industrial applications.

We differentiate between regression and classification trees. Regression trees are learned when the label is a quantitative and continuous variable (e.g., units sold). Classification trees are used when the label is a qualitative and categorical variable (e.g. item category).

Example 8.1. Recall the retail forecasting scenario for Example 6.4. The aim is to predict future sales over a training dataset D that is defined by a feature extraction query which has the following features: price and color of the item, and size and city of the store. The label is the number of units sold for the items and at different stores.

Figure 8.1 shows an example of a regression tree that could be used to predict the number of units sold. The inner nodes are conditional statements over the features. For a given tuple in D and inner decision tree node N, if the conditional statement at N is true, then continue to the left child of the node, otherwise continue to the right child. Leaf nodes give a prediction for the number of units sold.

Assume the data scientist wants to predict the number of units sold for a blue product at a small store (i.e. smaller than 100sqm) in Oxford. The regression tree would predict that 100 units are sold of this product, following the leftmost path in tree.

Classification trees could have the same structure, where leaf nodes replace numerical predictions with the most likely category in the domain of the label instead. □

Finding the optimal decision tree structure that best predicts the label for given the training dataset is an NP-hard problem [69]. Therefore, state-of-the-art decision tree learning algorithms use greedy procedures instead. This differentiates the learning of decision

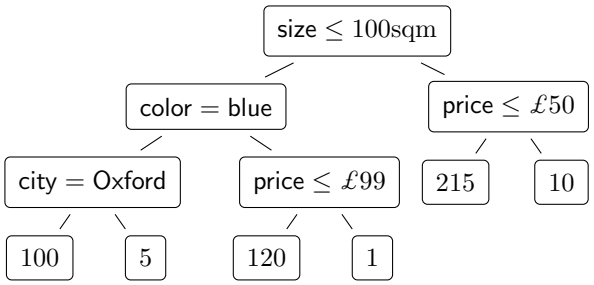


Figure 8.1: Regression tree used to predict sales for the retailer scenario from Example 8.1.

splitNode (dataset D_i, node N)

if (stopSplitting(N)) {
  if (regression) N.prediction $= \frac{1}{|D_i|} \sum_{(\boldsymbol{x},y)\in D_i} y$;
  if (classification) N.prediction $= \arg\max_{k\in \mathsf{Dom}(Y)} \sum_{(\boldsymbol{x},y)\in D_i} \mathbf{1}_{y=k}$;
  return;
}

$[X_j \text{ op } t_j] = \arg\min_{j\in[n]} \min_{t\in\mathcal{T}_j} \text{cost}(D_i, [X_j \text{ op } t_j]) + \text{cost}(D_i, [X_j \text{ !op } t_j])$;

N.condition $= [X_j \text{ op } t_j]$;
spitNode($\sigma_{X_j \text{ op } t_j} D_i, N$.leftChild);
splitNode($\sigma_{X_j \text{ !op } t_j} D_i, N$.rightChild);

Algorithm 8.1: Pseudo code for main procedure of the CART algorithm. Dataset D_i is a partition of the training dataset D with variables $X_1, \ldots X_n$ and label Y, and $\mathcal{T}_j$ denotes the set of thresholds for variable X_j.

trees from the classes of machine learning models that were considered in the previous two chapters, which learn the model parameters with a global optimization algorithm. In this chapter, we learn decision trees with the seminal CART algorithm [28], and show that for learning over feature extraction queries the algorithm reduces to computing many aggregates over the input database. Alternative decision tree learning algorithms, such as ID3 [110] or C4.5 [111], are similar to the CART algorithm and are also amenable to a database-centric solution.

The results presented in this chapter have previously been published in [118], and the background was adapted from [28, 92].

8.1 CART Algorithm

The CART algorithm [28] is a greedy algorithm, which recursively constructs the tree one node at a time. Algorithm 8.1 presents the pseudocode for the **splitNode** function, which

is the main procedure of the CART algorithm.

The algorithm takes as input a dataset D with variables $X_1, \ldots, X_n$ and label Y, as well as a decision tree node N. The core of the algorithm evaluates a predefined cost function over Y. The function takes as input the dataset D and a new threshold condition $[X_j \text{ op } t_j]$, and returns a scalar that represents the error of this condition over the provided dataset. The cost function is computed for all conditions $(X_j \text{ op } t_j)$, $\forall j \in [n], t_j \in \mathcal{T}_j$. The aim is to find the condition $(X_j \text{ op } t_j)$ that partitions the dataset D, so that the error on the resulting dataset partitions is minimized. For a continuous variable X_j, t_j is a real number and op is inequality (e.g., $\mathsf{age} < 25$). For a categorical variable X_j, t_j may be a subset of categories in $\mathsf{Dom}(X_j)$ and op denotes inclusion (e.g., $\mathsf{city} \in \{\mathsf{London}, \mathsf{Oxford}\}$). We use !op to denote the negation of op, i.e. if op denotes $\leq$ then !op denotes $>$.

Once this condition is found, a new node $\boxed{X_j \text{ op } t}$ is constructed and the algorithm proceeds recursively for each child. The input dataset for the left child is given by the dataset partition $\sigma_{X_j \text{ op } t} D$, and for the right child it is $\sigma_{X_j \text{ !op } t} D$.

The set of possible thresholds for variable X_j depends on the type of the variable. If X_j is a continuous variable, then $\mathcal{T}_j$ is given by the active domain $\mathsf{Dom}(X_j)$. In practice, it is common to restrict the size of $\mathcal{T}_j$ to 20-100 thresholds, which can be precomputed as percentiles over variable X_j in D. If X_j is a categorical variable, then $\mathcal{T}_j$ could contain all possible subsets of the categories in $\mathsf{Dom}(X_j)$, which implies that $|\mathcal{T}_j| = 2^{\mathsf{Dom}(X_j)}$. In practice, it is common to restrict $\mathcal{T}_j$ in one of two ways. A simple approach considers only conditions X_j op t_j where t_j is a category, and op denotes equality, such that the dataset is partitioned on one category versus all others. The second approach computes the cost for the split for each category, sorts the categories on their cost, and then seeks the partitioning on the ordered categories which minimizes the overall cost of the split [67]. This latter strategy has been shown to be an optimal partitioning of categorical variable for decision trees [39]. Note that both strategies for categorical variables require the evaluation of the cost function for each individual category.

The stopSplitting condition in Algorithm 8.1 restricts the complexity of the model to avoid overfitting on the training dataset. This is commonly done by restricting the size of the tree by specifying (1) the maximum depth of the tree, (2) number of instances required to continue splitting, or (3) minimum reduction in the cost required to split the node. These conditions are typically defined as input parameters. If the stopSplitting condition holds, then node N is a leaf node. The prediction of N depends on the type of tree that is learned. We next introduce regression and classification trees, and show how their data-intensive computation can be expressed as aggregate queries of the form (4.1).

8.2 Regression Trees

Regression trees are learned when the label Y is a continuous, quantitative variable. We assume that the training dataset D is defined by a feature extraction query Q over database instance I with relations $R_1, \ldots, R_m$.

Given node N in the decision tree, let X_i op t_i encode a potential condition for N, and $\mathcal{C}_N$ be the set of all conditions along the path from the root to N. Let $D_i = \sigma_{\mathcal{C}_N \cup \{X_i \text{ op } t_i\}} D$ denote a partition of the dataset D that satisfies all conditions $\mathcal{C}_N \cup \{X_i \text{ op } t_i\}$.

Example 8.2. For the tree depicted in Figure 8.1, the dataset fragment that is satisfied by the leftmost leaf node is given by: $D_i = \sigma_{\mathsf{size} \leq 100 \wedge \mathsf{color}=\mathsf{blue} \wedge \mathsf{city}=\mathsf{oxford}}(D)$. □

Let $\mu = \frac{1}{|D_i|} \sum_{(\mathbf{x},y) \in D_i} y$ denote the average over the label over dataset fragment D_i. At a leaf node, the prediction of a regression tree is given by μ.

The cost function for regression trees is given by the variance of the label Y over the dataset fragment D_i:

$$\begin{aligned} \text{variance} &= \sum_{(\boldsymbol{x},y) \in D_i} (y - \mu)^2 \\ &= \sum_{(\boldsymbol{x},y) \in D_i} y^2 - \frac{1}{|D_i|} \Big(\sum_{(\boldsymbol{x},y) \in D_i} y \Big)^2 \end{aligned} \tag{8.1}$$

Equation (8.1) can be computed as three aggregates $\texttt{SUM(1)}$, $\texttt{SUM}(Y)$, and $\texttt{SUM}(Y^2)$ over D_i. In the query language from Section 4.2, we can represent all three aggregates with the following query:

$$\mathsf{RTree}(\texttt{SUM(1)}, \texttt{SUM}(Y), \texttt{SUM}(Y^2)) = R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}), X_i \text{ op } t_i, \varphi(\omega_\varphi) \tag{8.2}$$

where $\varphi(\omega_\varphi)$ encodes the list of conditions in $\mathcal{C}_N$: $(X_j \text{ op } t_j)_{(X_j \text{ op } t_j) \in \mathcal{C}_N}$.

Recall that for categorical variable X_i, the CART algorithm considers each category $t_i \in \mathsf{Dom}(X_i)$ as a potential threshold. This translates into one query of form (8.2) for each $t_i \in \mathsf{Dom}(X_i)$, where each query has a selection condition $X_i = t_i$. The results of all these queries is equivalent to a single query which keeps X_i as a group by variable. Thus, for categorical variable X_i, we would compute the following query instead of (8.2):

$$\mathsf{RTree}(X_i, \texttt{SUM(1)}, \texttt{SUM}(Y), \texttt{SUM}(Y^2)) = R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}), \varphi(\omega_\varphi) \tag{8.3}$$

Queries (8.2) and (8.3) encode the conditions as selection conditions. An equivalent formulation for these queries encodes the conditions as a product of unary functions, where each function is an indicator function $\mathbf{1}_E$ that is equal to 1 if E is true and 0 otherwise.

For continuous variable X_j, this query is given by:

$$\mathsf{RTree}(\texttt{SUM}(\alpha), \texttt{SUM}(Y \cdot \alpha), \texttt{SUM}(Y^2 \cdot \alpha)) = R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}) \quad (8.4)$$
$$\text{where} \quad \alpha = \mathbf{1}_{X_i \text{ op } t_i} \cdot \prod_{[X_j \text{ op } t_j] \in \mathcal{C}_N} \mathbf{1}_{X \text{ op } t}$$

For categorical variable X_i, the equivalent encoding for (8.3) without selection conditions is given by:

$$\mathsf{RTree}(X_i, \texttt{SUM}(\alpha), \texttt{SUM}(Y \cdot \alpha), \texttt{SUM}(Y^2 \cdot \alpha)) = R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}) \quad (8.5)$$
$$\text{where} \quad \alpha = \prod_{[X_j \text{ op } t_j] \in \mathcal{C}_N} \mathbf{1}_{X \text{ op } t}$$

In the second encoding, all queries for the cost functions at node N share the same body. The advantage is that these queries can share computation across all queries that are computed for the decision tree node N. In Part II of this thesis, we assume that the queries are encoded as shown in (8.4) and (8.5).

Complexity Analysis

For each variable X_j, let c_j denote the number of queries that are computed for each split. If X_j is a continuous variable, then $c_j = |\mathcal{T}_j|$. If X_j is a categorical variable, then $c_j = 1$. We can now give the runtime for learning regression trees over feature extraction queries.

Theorem 8.3. *Let Q be a feature extraction query with n variables that is computed over input database I, where all relations in I have size at most N. Let $\mathcal{H} = (\mathcal{V}, \mathcal{E} = \mathcal{E}_R \cup \mathcal{E}_\infty)$ denote the query hypergraph of Q.*

Each node in a regression tree that is learned over Q can be computed in time:

$$O\left(|\mathcal{V}|^2 \cdot |\mathcal{E}| \cdot \sum_{j \in [n]} c_j \cdot N^{\mathsf{fhtw}(\mathcal{H})} \cdot \log N\right) \quad (8.6)$$

where $\mathsf{fhtw}(\mathcal{H})$ is the fractional hypertree width of $\mathcal{H}$, and c_j denotes the number of queries that need to be computed for variable X_j.

Proof. We have shown above that the data-intensive computation for each node in the regression tree is captured by queries (8.4) and (8.5), which are queries of the form (4.1).

Theorem 5.15 states that any query Q of the form (4.1) with free variables F can be computed in time $O\left(|\mathcal{V}|^2 \cdot |\mathcal{E}| \cdot s \cdot (N^{\mathsf{fhtw}_F(\mathcal{H})} + |Q|) \cdot \log N\right)$, where s is the number of products in the aggregate expression α.

For queries (8.4) and (8.5), we know that (1) they have at most one free variable, which means any tree decomposition is trivially F-connex and $\mathsf{fhtw}_F = \mathsf{fhtw}$; (2) following Proposition 5.20, we know that $|Q| = O(N)$, and thus $|Q|$ is always upper bounded by the

time it takes to evaluate the query; and (3) each aggregate is defined by product of unary functions and thus $s = 1$.

Finally, finding the optimal split for a given regression tree node can be done in one linear pass over the result of the queries that define the data-intensive computation, which is always less than the time it takes to compute these queries. □

We assume, without loss of generality, that a regression tree has a constant number of nodes. A remarkable fact that follows from Theorem 8.3 and Proposition 5.20, is that, for infinitely many feature extraction queries, we can learn the entire regression tree over a feature extraction query in time sub-linear in $|D|$ and thus faster than computing the result of the feature extraction query.

8.3 Classification Trees

For classification trees, the label Y has a set $\mathsf{Dom}(Y) = \{k_1, \ldots, k_p\}$ of categories. There exist many different cost functions that can be used to determine the optimal split condition for a node. Two commonly used cost functions are given by the entropy or Gini index:

$$\text{entropy} = -\sum_{k \in \mathsf{Dom}(Y)} \pi_k \log(\pi_k)$$
$$\text{gini} = 1 - \sum_{k \in \mathsf{Dom}(Y)} \pi_k^2$$

The aggregates π_k compute the frequencies of each category $k \in \mathsf{Dom}(Y)$ in the dataset fragment D_i. For category k, this frequency is the fraction of the tuples in D_i where $Y = k$ and of all tuples in D_i:

$$\pi_k = \frac{1}{|D_i|} \sum_{(\boldsymbol{x}, y) \in D_i} \mathbf{1}_{y=k} \tag{8.7}$$

For continuous variable X_i and condition $[X_i \text{ op } t_i]$, these frequencies can all be computed with the following two aggregate queries:

$$\mathsf{CTree}_1(Y, \texttt{SUM(1)}) = R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}), X_i \text{ op } t_i, \varphi(\omega_\varphi) \tag{8.8}$$
$$\mathsf{CTree}_2(\texttt{SUM(1)}) = R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}), X_i \text{ op } t_i, \varphi(\omega_\varphi) \tag{8.9}$$

where $\varphi(\omega_\varphi)$ encodes the list of conditions in $\mathcal{C}_N$: $(X_j \text{ op } t_j)_{(X_j \text{ op } t_j) \in \mathcal{C}_N}$.

For categorical variable X_i, we can instead compute one query which computes the cost

for all conditions $(X_i = t)_{t \in \mathsf{Dom}(X_i)}$ by setting X_i as a group-by variable:

$$\mathsf{CTree}_1(X_i, Y, \mathtt{SUM(1)}) = R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}), \varphi(\omega_\varphi) \tag{8.10}$$

$$\mathsf{CTree}_2(X_i, \mathtt{SUM(1)}) = R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}), \varphi(\omega_\varphi) \tag{8.11}$$

As for regression trees, there exists an equivalent encoding of these queries, which treats the selection conditions as unary indicator functions. For continuous variable X_j, the equivalent queries for (8.8) and (8.9) queries are given by:

$$\mathsf{CTree}_1(Y, \mathtt{SUM}(\alpha)) = R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}) \tag{8.12}$$

$$\mathsf{CTree}_2(\mathtt{SUM}(\alpha)) = R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}) \tag{8.13}$$

$$\text{where} \quad \alpha = \mathbf{1}_{X_i \text{ op } t_i} \cdot \prod_{[X_j \text{ op } t_j] \in \mathcal{C}_N} \mathbf{1}_{X_j \text{ op } t_j}$$

For categorical attribute X_j, the equivalent queries for (8.10) and (8.11) are given by:

$$\mathsf{CTree}_1(X_i, Y, \mathtt{SUM}(\alpha)) = R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}) \tag{8.14}$$

$$\mathsf{CTree}_2(X_i, \mathtt{SUM}(\alpha)) = R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}) \tag{8.15}$$

$$\text{where} \quad \alpha = \prod_{[X_j \text{ op } t_j] \in \mathcal{C}_N} \mathbf{1}_{X_j \text{ op } t_j}$$

Complexity Analysis

For each variable X_j, let c_j denote the number of queries that are computed for each split. If X_j is a continuous variable, then $c_j = |\mathcal{T}_j|$. If X_j is a categorical variable, then $c_j = 1$. We can now give the runtime for learning classification trees over feature extraction queries.

Theorem 8.4. *Let Q be a feature extraction query with n variables that is computed over input database I, where all relations in I have size at most N. Let $\mathcal{H} = (\mathcal{V}, \mathcal{E} = \mathcal{E}_R \cup \mathcal{E}_\infty)$ denote the query hypergraph of Q.*

Each node in a classification tree can be computed over Q in in time:

$$O\left(|\mathcal{V}|^2 \cdot |\mathcal{E}| \cdot \sum_{j \in [n]} c_j \cdot N^{\mathsf{fhtw}(\mathcal{H})+1} \cdot \log N\right) \tag{8.16}$$

where $\mathsf{fhtw}(\mathcal{H})$ is the fractional hypertree width of $\mathcal{H}$, and c_j denotes the number of queries that need to be computed for variable X_j.

Proof. We have shown above that the data-intensive computation for each node in the classification tree is captured by queries (8.12), (8.13), (8.14) and (8.15), all of which are queries of the form (4.1).

Theorem 5.15 states that any query Q of the form (4.1) with free variables F can

be computed in time $O\left(|\mathcal{V}|^2 \cdot |\mathcal{E}| \cdot s \cdot (N^{\mathsf{fhtw}_F(\mathcal{H})} + |Q|) \cdot \log N\right)$, where s is the number of products in the aggregate expression α.

The aggregates in queries (8.12), (8.13), (8.14) and (8.15) are defined by one product of unary functions, and therefore $s = 1$. In addition, the queries can have at most two free variables, and thus, following Proposition 5.13, we know that $\mathsf{fhtw}_F(\mathcal{H}) \leq \mathsf{fhtw}(\mathcal{H}) + 1$. Also, following Proposition 5.20, $|Q| = O(N^2)$, and thus $|Q|$ is always upper bounded by the time it takes to evaluate the query.

Finally, finding the optimal split for a given classification tree node can be done in one linear pass over the result of the queries that define the data-intensive computation, which is always less than the time it takes to compute these queries. □

We assume, without loss of generality, that a classification tree has a constant number of nodes. From Theorem 8.4 and Proposition 5.20 we know that, for infinitely many feature extraction queries Q, we can learn the entire classification tree over Q query in time sub-linear in $|Q|$ and thus faster than computing the result of the feature extraction query.

Chapter 9

Unsupervised Machine Learning

In this chapter, we consider unsupervised machine learning models. In contrast to the supervised models introduced in the previous chapters, unsupervised models aim to identify interesting and useful patterns in a dataset without pre-existing labels. We consider the following four popular unsupervised learning techniques:

(1) *Principal component analysis*, which is a technique used for dimensionality reduction. It maps a high dimension datasets into a lower dimensional data so that the lower dimensional representation captures most of the variance in the original data.

(2) *K-means clustering*, which divides a datasets into k clusters of "similar" data points with respect to some distance measure.

(3) *Mutual information* of two discrete random variables, which has many applications in machine learning. We highlight the applications for feature selection, learning decision trees, and learning the structure of Bayesian networks using Chow-Liu trees [40].

(4) *Data cubes*, a classic problem in data warehousing. Data cubes are used for exploratory analysis of large high dimension datasets.

For each of these applications, we show that they can be computed efficiently by expressing their data-intensive computation as aggregate queries of the form (4.1).

9.1 Principal Component Analysis

Principal component analysis (PCA) [128] is a statistical technique that is used for dimensionality reduction. The goal is to reduce a high dimensional dataset to a lower dimensional space, such that most of the variance of the original data is captured in the lower dimensional representation. This is done by computing the top-K principal components of the data, which extract the maximum variance of the data.

Finding the top-K principal components of a dataset D is equivalent to computing the top-K eigenvectors $(\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_K)$ and the corresponding eigenvalues $(\boldsymbol{\lambda}_1, \ldots, \boldsymbol{\lambda}_K)$ of the (centered) covariance matrix of D. Once we computed the top-K eigenvectors $\boldsymbol{\theta}$, the projection of a data point $\boldsymbol{x} \in D$ onto the lower K-dimensional space is given by the inner product $\boldsymbol{x}^\top \boldsymbol{\theta}$, where $\boldsymbol{\theta} = (\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_K)$ denotes the matrix of eigenvectors.

PCA has many applications in exploratory data analysis, and it is used as preprocessing step for predictive models. In this case, the input data for the model is defined by the lower dimensional data representation derived with PCA. This can be beneficial, because it is often easier to learn models over lower dimensional data.

We next consider principal component analysis (PCA) over the training dataset defined by a feature extraction query. We focus on the problem of computing the top-K eigenvectors of the covariance matrix, and show that the solution to this problem requires similar computations as our solution for square loss problems in Chapter 6. For ease of exposition, we first consider the case where the variables in D are all continuous variables, and then generalize the solution to continuous and categorical variables.

The results in this section have previously been published in [8]. The background information is adapted from [128, 92].

9.1.1 PCA with Continuous Variables

Consider a dataset D with continuous variables $X_1, \ldots, X_n$ that is the result of a feature extraction query Q over a database instance I with relations $R_1, \ldots, R_m$.

Let $\boldsymbol{\mu} = \frac{1}{|D|} \sum_{\boldsymbol{x} \in D} \boldsymbol{x}$ be the vector of means for each variable in the feature extraction query. The centered covariance matrix of D is given by:

$$\boldsymbol{\Sigma}_1 = \frac{1}{|D|} \sum_{\boldsymbol{x} \in D} (\boldsymbol{x} - \boldsymbol{\mu})(\boldsymbol{x} - \boldsymbol{\mu})^\top.$$

The top-K eigenvectors $\boldsymbol{\theta} = (\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_K)$ and their eigenvalues $\boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_K)$ can be computed one at a time using the min-max theorem based on the Rayleigh quotient [128]:

$$\max_{\boldsymbol{\theta}_j \in \mathbb{R}^n} \min_{\lambda_j \in \mathbb{R}} \boldsymbol{\theta}_j^\top \boldsymbol{\Sigma}_j \boldsymbol{\theta}_j + \lambda_j (\|\boldsymbol{\theta}_j\|^2 - 1), \tag{9.1}$$

where $\boldsymbol{\Sigma}_j$ is the centered covariance matrix after extracting the variance of the first $j-1$ principal components.

We compute the optimal solution for the top eigenvector $\boldsymbol{\theta}_1$ using a gradient-based optimization algorithm, which optimizes the following objective function by alternating between performing gradient ascent with respect to $\boldsymbol{\theta}_1$ and gradient descent with respect to λ_1 until convergence:

$$J(\boldsymbol{\theta}_1, \lambda_1) = \boldsymbol{\theta}_1^\top \boldsymbol{\Sigma}_1 \boldsymbol{\theta}_1 + \lambda_1 (\|\boldsymbol{\theta}_1\|^2 - 1). \tag{9.2}$$

The gradient optimization steps can then be done with Algorithm 6.1, where the gradient of $J(\boldsymbol{\theta}_1, \lambda_1)$ for the two subproblems is given by:

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}_1} J(\boldsymbol{\theta}_1, \lambda_1) = \boldsymbol{\Sigma}_1 \boldsymbol{\theta}_1 - 2\lambda_1 \boldsymbol{\theta}_1 \tag{9.3}$$

$$\boldsymbol{\nabla}_{\lambda_1} J(\boldsymbol{\theta}_1, \lambda_1) = \|\boldsymbol{\theta}_1\|^2 - 1 \tag{9.4}$$

The subsequent eigenvectors are computed with the same optimization procedure but over an updated covariance matrix that removes the variance captured by all previously computed principal components. The iteration step assumes we already computed the covariance matrix $\boldsymbol{\Sigma}_l$, the eigenvector $\boldsymbol{\theta}_l$, and the eigenvalue λ_l for $l \in [K-1]$. The step then computes the eigenvector $\boldsymbol{\theta}_{l+1}$ and the eigenvalue λ_{l+1} over the covariance matrix

$$\boldsymbol{\Sigma}_{l+1} = \boldsymbol{\Sigma}_l - \lambda_l \boldsymbol{\theta}_l \boldsymbol{\theta}_l^\top. \tag{9.5}$$

As for the square-loss problems from Chapter 6, we can compute $\boldsymbol{\Sigma}_1$ once, and then compute the eigenvectors without scanning the data again. If the data is centered in a preprocessing step, then the computation of $\boldsymbol{\Sigma}_1$ for PCA is identical to the covar matrix for linear regression. If the data is not centered, we can compute the covariance matrix using the following reformulation:

$$\begin{aligned}
\boldsymbol{\Sigma}_1 &= \frac{1}{|D|} \sum_{\boldsymbol{x} \in D} (\boldsymbol{x} - \boldsymbol{\mu})(\boldsymbol{x} - \boldsymbol{\mu})^\top \\
&= \frac{1}{|D|} \sum_{\boldsymbol{x} \in D} \boldsymbol{x}\boldsymbol{x}^\top - \frac{2\boldsymbol{\mu}}{|D|} \sum_{\boldsymbol{x} \in D} \boldsymbol{x}^\top + \frac{1}{|D|} \sum_{\boldsymbol{x} \in D} \boldsymbol{\mu}\boldsymbol{\mu}^\top \\
&= \frac{1}{|D|} \sum_{\boldsymbol{x} \in D} \boldsymbol{x}\boldsymbol{x}^\top - 2\boldsymbol{\mu}\boldsymbol{\mu}^\top + \boldsymbol{\mu}\boldsymbol{\mu}^\top \\
&= \frac{1}{|D|} \sum_{\boldsymbol{x} \in D} \boldsymbol{x}\boldsymbol{x}^\top - \boldsymbol{\mu}\boldsymbol{\mu}^\top.
\end{aligned} \tag{9.6}$$

Thus, we can compute the covariance matrix by first computing the non-centered matrix $\boldsymbol{\Sigma}$ as proposed for linear regression models in Section 6.3, and then subtracting $\boldsymbol{\mu}\boldsymbol{\mu}^\top$ to center the data.

The gradient with respect to $\boldsymbol{\theta}$ for PCA requires the same computation over $\boldsymbol{\Sigma}_1$ as the gradient for linear regression models (see Equation (6.13) and Example 6.7 from Section 6.3).

9.1.2 PCA with Continuous and Categorical Variables

PCA is based on the analysis of variance between variables, and therefore it cannot be computed directly over categorical data. It can however be meaningful to compute PCA over one-hot encoded categorical data, which would provide insights into the variance of the frequency of the co-occurrence of categories for different categorical variables.

We can compute PCA over one-hot encoded categorical variables efficiently by computing it over the sparse representation of the covariance matrix, which is a variant of the representation of the $\mathbf{\Sigma}$ matrix that we introduced for the case of square-loss problems in Chapter 6. Recall our encoding for the feature vector from Section 6.1 that uniformly captures continuous and categorical variables. For a given tuple $\boldsymbol{x} \in D$, let $\boldsymbol{x} = (\boldsymbol{x}_c)_{c\in[n]}$ denote the feature vector of the model, where each component $\boldsymbol{x}_c$ is also a vector. If X_c is a categorical variable, the vector $\boldsymbol{x}_c = (\mathbf{1}_{X_c=v})_{v\in\mathsf{Dom}(X_c)}$ is the indicator vector derived from the one-hot encoding of X_c. If X_c is a continuous variable, $\boldsymbol{x}_c = (x_c)$ encodes a scalar. Similarly, we encode each eigenvector as a vector $\boldsymbol{\theta} = (\boldsymbol{\theta}_c)_{c\in[n]}$, where each component $\boldsymbol{\theta}_c$ is a vector of the same dimension as $\boldsymbol{x}_c$.

One difference between the square loss problems and PCA is that PCA requires its variables to be linearly independent. This property is not satisfied by one-hot encoding, because it is possible to derive the indicator value for one category based on a linear combination of the indicator values for all other categories. For this reason, it is common practice to do one-hot encoding of the categorical variables for all but one category. In our problem formulation, this means that for a categorical variable X_c, we encode the corresponding component $\boldsymbol{x}_c$ as an indicator vector whose size is the number of its categories minus one, so that we one-hot encode over all but the last category of X_c. This encoding is often referred to as *dummy encoding* in many data science tools.

Another difference is the requirement to center the data. For categorical data, it is not desirable to center the data in a preprocessing step, as this would require a one-hot encoding of the input relations. To avoid the materialization of the one-hot encoding, we compute the non-centered matrix first, and then subtract $\boldsymbol{\mu}\boldsymbol{\mu}^\top$, as shown in (9.6). The sparse representation of the covariance matrix is then a block matrix, where each entry $\boldsymbol{\sigma}_{ij} \in \mathbf{\Sigma}_1$ is defined as:

$$\boldsymbol{\sigma}_{ij} = \frac{1}{|D|}\sum_{\boldsymbol{x}\in D} \boldsymbol{x}_i\boldsymbol{x}_j^\top - \boldsymbol{\mu}_i\boldsymbol{\mu}_j^\top \tag{9.7}$$

The vector of means $\boldsymbol{\mu}$ has the same dimension as $\boldsymbol{x}$, where for each categorical variable X_c the component $\boldsymbol{\mu}_c \in \boldsymbol{\mu}$ is the vector of frequencies for all but one category in the $\mathsf{Dom}(X_c)$:

$$\boldsymbol{\mu}_c = \frac{1}{|D|}\sum_{\boldsymbol{x}\in D} \boldsymbol{x}_c \tag{9.8}$$

The vector $\boldsymbol{\mu}_c$ can be computed efficiently as a `COUNT` query with group-by variable X_c. We drop the group with the lowest count and divide the count for each other group by $|D|$.

The resulting matrix $\mathbf{\Sigma}_1$ has the same structure as the sparse tensor that is computed for linear regression problems. In fact, the quantity $\boldsymbol{\sigma}_{ij}$ in (9.7) is simply the centered variant of the expression in (6.12), which can be computed as Covar queries of the form (6.15), (6.16), or (6.17). The centering of the $\mathbf{\Sigma}_1$ as well as updating the matrix for subsequent eigenvectors

can be expressed as group-by aggregate queries and computed without materializing the products $\boldsymbol{\mu}\boldsymbol{\mu}^\top$ and $\boldsymbol{\theta}\boldsymbol{\theta}^\top$.

We consider the case where X_j and X_k are categorical variables. A sparse representation of the non-centered entry $\boldsymbol{\sigma}_{ij}$ of $\boldsymbol{\Sigma}$ is given by query $\mathsf{Covar}(X_j, X_k, \texttt{SUM(1)})$ from Equation (6.17). We can center $\boldsymbol{\sigma}_{ij}$ by subtracting $\boldsymbol{\mu}_j\boldsymbol{\mu}_k^\top$ with the following query:

$$\mathsf{Covar}_{ij}(X_j, X_k, \texttt{SUM(}C - P_1 \cdot P_2\texttt{)}) \leftarrow \mathsf{Covar}_{ij}(X_j, X_k, C), \boldsymbol{\mu}_j(X_j, P_1), \boldsymbol{\mu}_k(X_k, P_2) \quad (9.9)$$

Let $\boldsymbol{\theta}$ encode an eigenvector, where $\boldsymbol{\theta}_j$ and $\boldsymbol{\theta}_k$ are the components in $\boldsymbol{\theta}$ that correspond to variables X_j and X_k. Let λ encode the eigenvalue for $\boldsymbol{\theta}$. Then, we can update the centered $\boldsymbol{\sigma}_{ij}$ by $\boldsymbol{\theta}_j\boldsymbol{\theta}_k^\top$ with the following query:

$$\mathsf{Covar}_{ij}(X_j, X_k, \texttt{SUM(}C - \lambda \cdot P_1 \cdot P_2\texttt{)}) \leftarrow \mathsf{Covar}_{ij}(X_j, X_k, C), \boldsymbol{\theta}_j(X_j, P_1), \boldsymbol{\theta}_k(X_k, P_2) \quad (9.10)$$

The gradient (9.3) with respect to the eigenvector $\boldsymbol{\theta}$ requires the product of each entry $\boldsymbol{\sigma}_{ij}$ in the centered covariance matrix $\boldsymbol{\Sigma}$ and the corresponding component $\boldsymbol{\theta}_j$ of $\boldsymbol{\theta}$. We can compute this product with the following query (assuming X_i and X_j are categorical variables):

$$Q(X_j, \texttt{SUM(}C \cdot P\texttt{)}) \leftarrow \mathsf{Covar}_{ij}(X_j, X_k, C), \boldsymbol{\theta}_j(X_j, P) \quad (9.11)$$

where Covar_{ij} represents the centered entry in the covariance matrix $\boldsymbol{\Sigma}$. This query is identical to the query (6.18), which defines the data-intensive computation of the gradient for linear regression models.

We next exemplify the required computation for PCA using the retail forecasting scenario from Section 6.3.

Example 9.1. Consider the entry $\boldsymbol{\sigma}_{ij} \in \boldsymbol{\Sigma}_1$ where $i = (\mathsf{store})$ and $j = (\mathsf{city})$. We can compute the centered entry in the covariance matrix based on the non-centered entry $\boldsymbol{\sigma}_{ij}$ and the frequency vectors for store and city. The non-centered entry is computed with the following query which follows from the Equation (6.17):

$$\mathsf{Covar}_{(\mathsf{store},\mathsf{city})}(\mathsf{store}, \mathsf{city}, \texttt{SUM(1)}) \leftarrow R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m})$$

Let $\boldsymbol{\mu}_s(\mathsf{store}, \alpha_s)$ and $\boldsymbol{\mu}_c(\mathsf{city}, \alpha_c)$ be the relational encoding of the frequency vectors for store and respectively city. The relations store tuples that give for each store and respectively city the corresponding frequency that is denoted by α_s and respectively α_c. We can then compute the $\boldsymbol{\sigma}_{ij}^{(1)}$ entry in the centered covariance matrix $\boldsymbol{\Sigma}_1$ without materializing the product of $\boldsymbol{\mu}_s$ and $\boldsymbol{\mu}_c^\top$ with the following SQL query:

$$\boldsymbol{\sigma}_{ij}^{(1)}(\mathsf{store}, \mathsf{city}, \texttt{SUM(}c - \alpha \cdot \beta\texttt{)}) \leftarrow \mathsf{Covar}_{(\mathsf{store},\mathsf{city})}(\mathsf{store}, \mathsf{city}, c), \boldsymbol{\mu}_s(\mathsf{store}, \alpha), \boldsymbol{\mu}_c(\mathsf{city}, \beta)$$

Let $\boldsymbol{\theta}_s(\mathsf{store}, \alpha_s)$ and $\boldsymbol{\theta}_c(\mathsf{city}, \alpha_c)$ be the relational encodings of the components in $\boldsymbol{\theta}_1$ that

correspond to store and respectively city. We can compute the updated entry $\boldsymbol{\sigma}_{ij}^{(2)} \in \boldsymbol{\Sigma}_2$ based on (9.5) without materializing the product of $\boldsymbol{\theta}_s$ and $\boldsymbol{\theta}_c^\top$ with the following query:

$$\boldsymbol{\sigma}_{ij}^{(2)}(\mathsf{store}, \mathsf{city}, \texttt{SUM}(c - \lambda_1 \cdot \alpha \cdot \beta)) \leftarrow \boldsymbol{\sigma}_{(\mathsf{store},\mathsf{city})}^{(1)}(\mathsf{store}, \mathsf{city}, c), \boldsymbol{\theta}_s(\mathsf{store}, \alpha), \boldsymbol{\theta}_c(\mathsf{city}, \beta)$$

Finally, the data-intensive computation of the gradient with respect to eigenvector $\boldsymbol{\theta}$ requires the product of $\boldsymbol{\sigma}_{ij}^{(2)} \in \boldsymbol{\Sigma}_2$ and $\boldsymbol{\theta}_c^\top$, which can be computed with the following query:

$$Q(\mathsf{store}, \texttt{SUM}(c \cdot \alpha)) \leftarrow \boldsymbol{\sigma}_{(\mathsf{store},\mathsf{city})}^{(1)}(\mathsf{store}, \mathsf{city}, c), \boldsymbol{\theta}_c(\mathsf{city}, \alpha)$$

□

Complexity analysis

Theorem 9.2. *Let Q be a feature extraction query over with variables $X_1, \ldots, X_n$ that is computed over a database instance I where all relations have size at most N. Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ denote the query hypergraph of Q.*

We can compute the top-K eigenvectors for principal component analysis over Q in time:

$$O\left(|\mathcal{V}|^2 \cdot |\mathcal{E}| \cdot n^2 \cdot N^{\mathsf{fhtw}(\mathcal{H})+1} \cdot \log N + \xi \cdot K \cdot \sum_{i,j \in [n]} |\boldsymbol{\sigma}_{ij}|\right), \tag{9.12}$$

where ξ denotes the maximum number of batch gradient descent iterations performed to learn each of the K eigenvectors.

Proof. The first part of (9.12) captures the computation of the centered covariance matrix. We have shown above that the data-intensive computation of the non-centered covariance matrix is identical to the data-intensive computation for linear regression models. Also, we can center the matrix in time linear in $|\boldsymbol{\Sigma}| = O(n^2 \cdot N^{\mathsf{fhtw}(\mathcal{H})+1})$. The runtime bound for this part therefore follows directly from Theorem 6.8.

The second part of (9.12) captures the computation of the gradient and updating the covariance matrix. Queries (9.10) and (9.11) show that we can compute each of these steps in time linear in $|\boldsymbol{\Sigma}|$. The runtime bound follows from the fact that we compute one update and at most ξ gradients for each of the K eigenvectors. □

9.2 K-Means Clustering

In this section, we consider the popular k-means clustering algorithm. Given a dataset $D \subseteq \mathbb{R}^n$, the aim of clustering is to divide D into k clusters of "similar" data points $D = \bigcup_{i \in [k]} D_i$ where the similarity is defined with respect to the Euclidean distance and k is a given fixed positive integer [72]. Each cluster D_i is represented by a cluster mean

$\boldsymbol{\mu}_i \in \mathbb{R}^n$. The k-means problem is to find the cluster means $(\boldsymbol{\mu}_i)_{i\in[k]}$ that minimize the optimization problem:

$$\min_{(\boldsymbol{\mu}_1,\ldots,\boldsymbol{\mu}_k)} \sum_{\boldsymbol{x}\in D} \min_{i\in[k]} \left(\|\boldsymbol{x} - \boldsymbol{\mu}_i\|_2^2 \right) \tag{9.13}$$

Other norms or distance measures can be used, e.g., if we replace the ℓ_2-norm with ℓ_1-norm, then this becomes the k-median problem. In the following, we consider the ℓ_2-norm.

One of the most ubiquitous clustering methods is Lloyd's k-means clustering algorithm [82] (also known as the k-means method). Lloyd's algorithm can be viewed as a special instantiation of the *Expectation-Maximization* (EM) algorithm. It iteratively computes two updating steps until convergence. First, it updates the cluster assignments for each $(D_i)_{i\in[k]}$:

$$D_i = \left\{ \boldsymbol{x} \in D \mid \|\boldsymbol{x} - \boldsymbol{\mu}_i\|^2 \leq \|\boldsymbol{x} - \boldsymbol{\mu}_j\|^2, \forall j \in [k] \right\} \tag{9.14}$$

and then it updates the corresponding k-mean vectors $(\boldsymbol{\mu}_i)_{i\in[k]}$:

$$\boldsymbol{\mu}_i = \frac{1}{|D_i|} \sum_{\boldsymbol{x}\in D_i} \boldsymbol{x} \tag{9.15}$$

In the following, we consider the case where the dataset D is the result of a feature extraction query Q over database I with relations $R_1, \ldots, R_m$. We present two different approaches for solving the k-means problem over I without materializing D: (1) The first approach rewrites the update steps (9.14) and (9.15) into aggregate queries with additive inequalities, which can be computed directly over the database I. (2) The second approach uses a novel algorithm, called Rk-means, which approximates the k-means objective by clustering over a small coreset of D instead. We show that the data-intensive computation of Rk-means can be cast into aggregate queries of the form (4.1) (without additive inequalities).

The rewriting for both approaches shows that we are able to compute the solution for k-means asymptotically faster than the time it takes to materialize the feature extraction query. In practice, however, the two approaches face a tradeoff between performance and accuracy. In the first approach, we compute Lloyd's algorithm exactly, but this requires the computation of aggregate queries with inequalities that range over all variables. As discussed in Section 7.4, this essentially means that we need to pre-materialize two bags for the underlying tree decomposition, which potentially requires joining several relations. This is not required for the second approach, because the aggregate queries for Rk-means do not have additive inequalities. Rk-means, however, computes a constant factor approximation of the result, which can be larger than the approximation ratio of Lloyd's algorithm.

The first approach has been presented in [5], whereas the second approach was shown in [42]. The background information for this chapter was adapted from [72, 92].

9.2.1 K-Means Clustering via Aggregates with Inequalities

Our first approach to solve the k-means problem is based on the observation that we can reformulate the updating of the k means as aggregate queries with inequalities. In fact, we can update the k means without explicitly computing the partitioning $(D_i)_{i\in[k]}$. For ease of notation, we consider the case where all variables in D are continuous variables.

For a given set of k mean vectors $\{\boldsymbol{\mu}_l\}_{l\in[k]}$ and two integers $i, j \in [k]$, let $c_{ij}(\boldsymbol{x})$ be the following function:

$$\begin{aligned} c_{ij}(\boldsymbol{x}) &= \sum_{\ell\in[n]} [(x_\ell - \mu_{i,\ell})^2 - (x_\ell - \mu_{j,\ell})^2] \\ &= \sum_{\ell\in[n]} [\mu_{i,\ell}^2 - 2x_\ell(\mu_{i,\ell} + \mu_{j,\ell}) - \mu_{j,\ell}^2], \end{aligned} \tag{9.16}$$

where $\mu_{j,\ell}$ is the ℓ'th component of mean vector $\boldsymbol{\mu}_j$. A tuple $\boldsymbol{x} \in D$ is closest to center $\boldsymbol{\mu}_i$ if and only if $c_{ij}(\boldsymbol{x}) \leq 0$ holds $\forall j \in [k]$.

Note that $c_{ij}(\boldsymbol{x}) \leq 0$ encodes an additive inequality of the form (4.2). We use this inequality to reformulate the computation of the mean vector $\boldsymbol{\mu}_i$ as an aggregate query with additive inequalities of the form (4.1):

$$Q_i(\texttt{SUM(1)}, \texttt{SUM}(X_1), \ldots, \texttt{SUM}(X_n)) \leftarrow R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}), (c_{ij}(\mathcal{V}) \leq 0)_{j\in[n]} \tag{9.17}$$

where the first aggregate computes $|D_i|$, and the remaining aggregates compute the sum $\sum_{\boldsymbol{x}\in D_i} x_\ell$ for each $\ell \in [n]$. We can then define the mean vector $\boldsymbol{\mu}_i$ as follows:

$$\boldsymbol{\mu}_i(\frac{s_1}{c}, \ldots, \frac{s_n}{c}) \leftarrow Q_i(c, s_1, \ldots, s_n) \tag{9.18}$$

Overall, each update step in the Lloyd's algorithm can be computed with $O(n)$ aggregate queries with inequalities of the form (4.1).

Complexity analysis

Theorem 9.3. *Let Q be a feature extraction query with n (continuous) variables that is computed over a database $\boldsymbol{I}$, where each relation in I has size at most N. Let $\mathcal{H} = (\mathcal{V}, \mathcal{E} = \mathcal{E}_R \cup \mathcal{E}_\infty)$ denote the query hypergraph of Q.*

Each iteration of Lloyd's k-means algorithm can be computed in time:

$$O(|\mathcal{V}|^2 \cdot |\mathcal{E}| \cdot n^2 \cdot N^{\mathsf{r\text{-}fhtw}(\mathcal{H})} \log^{k-1} N)$$

Proof. We have shown that each mean vector $(\boldsymbol{\mu}_j)_{j\in[k]}$ can be computed with one aggregate query of the form (4.1), which has $|\mathcal{E}_\infty| = k$ additive inequalities and no group-by variables.

The overall runtime to update all k-means follows from Theorem 5.18, which states a

query φ of the form (4.1) with one aggregate that is defined by a single product of functions ($s = 1$) and no free variables can be computed in time $O(|\mathcal{V}|^2 \cdot |\mathcal{E}| \cdot N^{\mathsf{r\text{-}fhtw}(\mathcal{H})} \cdot \log^{k-1} N)$. □

9.2.2 K-Means Clustering via Approximation with Coresets

We next present an alternative algorithm for k-means clustering, called Rk-means (or *relational k-means*) [42], which approximates the k-means objective by computing the k clusters over a small coreset of D. A coreset of D is a small set of points that provide a good summarization of the original dataset D. Rk-means constructs a so-called *grid coreset*, which is defined as the Cartesian product of clusters computed over disjoint projections of D. Rk-means gives a constant approximation for the k-means objective, and asymptotic runtime improvements over the mainstream approach of clustering over the materialized feature extraction query.

We consider the case where the dataset D is defined by a feature extraction query Q with n variables over a database with relations $R_1, \ldots, R_m$. Rk-means efficiently supports continuous and categorical variables, where the latter are one-hot encoded. To explain the algorithm, we assume, without loss of generality, that all categorical variables in D are one-hot encoded. We then explain how Rk-means can efficiently accommodate for categorical variables without explicitly one-hot encoding them. We use d to denote the dimensionality of the D after one-hot encoding.

Before we describe the algorithm, we first define the weighted k-means problem.

Definition 9.4. A *weighted k-means instance* is a pair (D, w), where D is a set of points in $\mathbb{R}^n$ and $w : D \to \mathbb{R}^+$ is a weight function. The task is to find a set $\boldsymbol{C} = \{\boldsymbol{\mu}_1, \cdots, \boldsymbol{\mu}_k\}$ of k centroids to solve the optimization problem:

$$\min_{(\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k)} \sum_{\boldsymbol{x} \in D} w(\boldsymbol{x}) \cdot \min_{i \in [k]} \|\boldsymbol{x} - \boldsymbol{\mu}_i\|_2^2 \tag{9.19}$$

□

Algorithm 9.1 presents the pseudocode of Rk-means. The algorithm takes as input the query Q, and two constants k and κ, which define the number of clusters used by Rk-means, and an arbitrary partitioning $S_1 \cup \cdots \cup S_p$ of the dimensions $[d]$ into non-empty subsets. The algorithm then clusters the dataset D in four steps.

Step 1 For each $j \in [p]$, Rk-means projects D onto the subspace S_j and computes the total weight for each point in the projection. For a given point $\boldsymbol{x}_{S_j} \in \pi_{S_j} D$, the weight is given by $|\sigma_{S_j = \boldsymbol{x}_{S_j}} D|$, i.e., the number of points in the partition of D that has values $\boldsymbol{x}_{S_j}$ for variables S_j. The projection onto the variables S_j and the corresponding weight can be computed with the following `COUNT` query of the form (4.1):

$$Q_j(S_j, \texttt{SUM(1)}) \leftarrow R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}) \tag{9.20}$$

Rk-means(Query Q, Number of clusters k, $\kappa \geq 2$, Partitioning $[d] := S_1 \cup \cdots \cup S_p$)
let : $Q(\mathcal{V}) \leftarrow R_1(\omega_{R_1}), \ldots, \mathbb{R}_m(\omega_{R_m})$ where $\mathcal{V} = [n]$;

foreach $j \in [p]$ {
  $Q_j(S_j, \texttt{SUM(1)}) \leftarrow R_1(\omega_{R_1}), \ldots, \mathbb{R}_m(\omega_{R_m})$; /* Step 1 */
  $C_j(S_j, c_j) \leftarrow \mathsf{wkmeans}_1(Q_j, \kappa)$; /* Step 2 */
}

$G(c_1, \ldots, c_p, \texttt{SUM(1)}) \leftarrow R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}), C_1(S_1, c_1), \ldots, C_p(S_p, c_p)$; /* Step 3 */
$C(c_1, \ldots, c_m, c) \leftarrow \mathsf{wkmeans}_2(G, k)$; /* Step 4 */

return $\pi_c C$;

Algorithm 9.1: Pseudocode of the Rk-means algorithm. The algorithm has four steps, which (1) project the result of Q onto subspaces $S_1 \cup \cdots \cup S_p$, (2) cluster each subspace with a weighted k-means algorithm $\mathsf{wkmeans}_1$ to obtain κ clusters, (3) construct a coreset G over the clustered subspaces, and (4) perform weighted k-means with algorithm $\mathsf{wkmeans}_2$ on the coreset G. The coreset G is constructed without materializing the result of Q.

Step 2 For each $j \in [p]$, we then perform weighted k-means to obtain κ individual clusters on each subspace S_j for some $\kappa \geq 2$. These are solved using some weighted k-means algorithm denoted by $\mathsf{wkmeans}_1$ over the query Q_j from Step 1. In Algorithm 9.1, we assume that $\mathsf{wkmeans}_1$ returns a "cluster assignment" relation C_j which records for each $\boldsymbol{z} \in \pi_{S_j}(D)$ the corresponding closest centroid c_j in the S_j subspace.

Step 3 Then, using the results of these clusterings we assemble a cross-product weighted grid G of centroids, which defines the coreset of D. A grid point $\boldsymbol{g}$ in the coreset is composed of tuples of size p, with the value in dimension $j \in [p]$ ranging over the possible cluster means for that subspace computed in Step 2. The weight of a grid point $\boldsymbol{g} = (c_1, \ldots, c_p)$ is the number of data points in D closest to the grid point. Since the ℓ_2^2-distance decomposes into a sum over ℓ_2^2-distances on each subspaces, an input data point $\boldsymbol{x} \in D$ is closest to $\boldsymbol{g}$ if and only if the projections of $\boldsymbol{x}$ onto each subspace S_j is closest to c_j. The grid coreset G as well as the weight for each grid point $\boldsymbol{g}$ can be computed with the aggregate query:

$$G(c_1, \ldots, c_p, \texttt{SUM(1)}) \leftarrow R_1(\omega_{R_1}), \ldots, \mathbb{R}_m(\omega_{R_m}), C_1(S_1, c_1), \cdots, C_p(S_p, c_p) \tag{9.21}$$

Step 4 Finally, we perform weighted k-means clustering on the coreset G using the algorithm denoted $\mathsf{wkmeans}_2$ to reduce down to the desired result of k centroids.

Step 2 and Step 4 use different algorithms, because it is possible to use optimal clustering algorithms for certain subspaces. We use a different number of clusters for the two steps to limit the size of the coreset G. Typically, one would take $\kappa \leq k$.

Approximation analysis Let $\mathsf{wkmeans}_1$ and $\mathsf{wkmeans}_2$ have approximation ratios α and γ. Then, the approximation ratio of Rk-means is $(\sqrt{\gamma}+\sqrt{\alpha}+\sqrt{\gamma\alpha})^2$ [42]. In particular, we can define subspaces $S_1 \cup \ldots \cup S_p$ that guarantee that $\alpha = 1$, so that the overall approximation ratio is $(1+2\sqrt{\gamma})^2$. If both sub-problems are solved optimally ($\alpha = \gamma = 1$), then the solution of Rk-means is a 9-approximation. For comparison, the best known approximation ratio is 6.357 for data in Euclidean space [13].

Subspaces for optimal clustering

The approximation ratio for Step 2 highly depends on the choice of the subspaces $S_1 \cup \ldots \cup S_p$ of dimensions $[d]$. We next show that we can choose subspaces, so that we can compute an optimal clustering of the subspace, and thus $\alpha = 1$. The subspaces also avoid the explicit one-hot encoding for categorical variables in the input data.

Recall that the feature extraction query Q has n variables $X_1, \ldots, X_n$, which are mapped into d features after one-hot encoding. A straightforward choice for the subspaces is to choose one subspace S_i for each variable X_i in the feature extraction query.

If X_i is a continuous variable, then the subspace S_i defines a one-dimensional weighted k-means problem, which can be solved optimally with dynamic programming [134].

If X_i is a categorical variable, then the subspace S_i is multi-dimensional, and contains the one-hot encoded indicator vectors over the domain of X_i. We call this subspace a *categorical subspace*.

Consider a weighted k-means subproblem solved by $\mathsf{wkmeans}_1$ defined on a categorical subspace induced by a categorical feature X_i that has L categories. Then, the instance is of the form $(\boldsymbol{I}, w)$, where $\boldsymbol{I}$ is the collection of L indicator vectors $\mathbf{1}_e$, one for each element $e \in \mathsf{Dom}(X_i)$ (One can think of $\boldsymbol{I}$ as the identity matrix of order L), and $w : \mathbf{1} \to \mathbb{R}^+$ is a weight function.

Theorem 9.5 (Adapted from [42])**.** *Given a categorical weighted k-means instance $(\boldsymbol{I}, w)$, an optimal solution can be obtained by putting each of the $k-1$ highest weight indicator vectors in its own cluster, and the remaining vectors in the same cluster.*

Theorem 9.5 shows that we can construct the optimal solution for a single categorical dimension in the time it takes to find the number of points in each category. In fact, Rk-means can compute this clustering without explicitly one-hot encoding X_i: (1) Step 1 projects the data matrix D onto the variable X_i, and computes the corresponding weight with query (9.20); and (2) Step 2 sorts the result of (9.20) in decreasing order with respect to the weight, where the heaviest $k-1$ elements form their own centroid, while the remaining vectors are clustered together (the "light cluster"). Step 2 thus takes time $O(L \log L)$ for a categorical subspace of dimension L.

We next present how we can also accelerate the computation of the clustering in Step 4 without explicitly one-hot encoding the categorical variable X_i.

Accelerating the implementation of Rk-means

We next discuss optimizations and accelerations of Step 4 of the Rk-means algorithm, where the categorical subspace kmeans problem is solved trivially using Theorem 9.5.

Let S_j denote is a categorical subspace corresponding to a categorical variable K where $\mathsf{Dom}(K) = \{e_1, \ldots, e_L\}$. Without loss of generality, assume $w(\mathbf{1}_{e_1}) \geq \cdots \geq w(\mathbf{1}_{e_L})$, then a heavy centroid $i \in [k-1]$ is represented by $\mathbf{1}_{e_i}$, and the centroid of the light cluster is an L-dimensional vector $\boldsymbol{c} = (s_e)_{e \in \mathsf{Dom}(K)}$, where

$$s_{e_i} := \begin{cases} 0 & i \in [k-1] \\ \frac{w(\mathbf{1}_{e_i})}{\sum_{j=k}^{L} w(\mathbf{1}_{e_j})} & i \geq k \end{cases} \tag{9.22}$$

This encoding is sound and space-inefficient.

Remember also that Step 4 clusters the coreset G using a modified version of Lloyd's weighted k-means that exploits the structure of G and sparse representation of categorical values. We show how to improve the distance computation $\|\boldsymbol{c}_j - \boldsymbol{\mu}_j\|^2$ for sub-space S_j, where $\boldsymbol{c}_j$ and $\boldsymbol{\mu}_j$ are the j-th components of a grid point and respectively of a centroid for G. Since this subspace corresponds to a categorical variable K with, say, L_j categories, it is mapped into L_j sub-dimensions. Let $\boldsymbol{c}_j = [s_1, \ldots, s_{L_j}]$ and $\boldsymbol{\mu}_j = (t_1, \ldots, t_{L_j})$. Using the explicit one-hot encoding of its categories, we would need $O(L_j)$ time to compute $\|\boldsymbol{c}_j - \boldsymbol{\mu}_j\|^2 = \sum_{\ell \in [L_j]} (s_\ell - t_\ell)^2$. We can instead achieve $O(1)$ time: there are k distinct values for $\boldsymbol{c}_j$ by our coreset construction, each represented by a vector of size L_j with one non-zero entry for $k-1$ of them and $L_j - k + 1$ non-zero entries for one of them.

If $\boldsymbol{c}_j = \mathbf{1}_e$ is an indicator vector for some element $e \in K$ (e is one of the $k-1$ heavy categories), then

$$\|\boldsymbol{c}_j - \boldsymbol{\mu}_j\|^2 = \|\mathbf{1}_e - \boldsymbol{\mu}_j\|^2 = 1 - 2t_e + \|\boldsymbol{\mu}_j\|^2 . \tag{9.23}$$

If $\boldsymbol{c}_j$ is a light cluster centroid,

$$\|\boldsymbol{c}_j - \boldsymbol{\mu}_j\|^2 = \|\boldsymbol{c}_j\|^2 + \|\boldsymbol{\mu}_j\|^2 - 2\langle \boldsymbol{c}_j, \boldsymbol{\mu}_j \rangle . \tag{9.24}$$

In (9.23), by pre-computing $\|\boldsymbol{\mu}_j\|^2$ we only spend $O(1)$-time per heavy element e. In (9.24), by also pre-computing $\|\boldsymbol{c}_j\|^2$ and $\langle \boldsymbol{c}_j, \boldsymbol{\mu}_j \rangle$, and by noticing that $\boldsymbol{c}_j$ is $(L_j - k + 1)$-sparse, the time spent for this step is $O(L_j - k)$. Overall, we spend time $O(L_j)$ for computing $\|\boldsymbol{c}_j - \boldsymbol{\mu}_j\|^2$ per categorical dimension, modulo the precomputation time.

Step 4 thus requires $O(|G|mk + \sum_{j \in [m]} L_j k) = O(|G|mk + Nkm)$ per iteration, whereas a generic approach would take time $O(\sum_{j \in [m]} |G| k L_j) = O(|G| k N m)$. Our modified weighted k-means algorithm thus saves time proportional to the domain sizes of the categorical variables, which may be as large as N.

Complexity analysis

Theorem 9.6. *Let Q denote a feature extraction query that is computed over a relational database I, where each relation in I has size at most N. Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ denote the query hypergraph of Q. Let n be number of free variables in Q.*

Rk-means computes the k cluster means of the data matrix D defined by Q in time:

$$O(|\mathcal{V}|^2 \cdot |\mathcal{E}| \cdot \min(N^{\mathsf{fhtw}(\mathcal{H})} k^n, N^{\rho^*_{\mathcal{H}}(\mathcal{V})}) \log N + k \cdot n \cdot N^2) \tag{9.25}$$

using a coreset of D with size $O(\min\{k^n, N^{\rho^*_{\mathcal{H}}(\mathcal{V})}\})$.

Proof. We analyze the time complexity for each of the four steps of the Rk-means algorithm.

Step 1 computes one query (9.20) for each group-by variable of Q. Following Theorem 5.15, this can be computed in time $O(|\mathcal{V}|^2 \cdot |\mathcal{E}| \cdot N^{\mathsf{fhtw}} \cdot \log N)$, with an overall runtime of $O(|\mathcal{V}|^2 \cdot |\mathcal{E}| \cdot n \cdot N^{\mathsf{fhtw}} \cdot \log N)$.

In Step 2, the optimal clustering in each dimension takes time $O(N \log N)$ for each categorical and $O(k \cdot N^2)$ for each continuous variable, with an overall runtime of $O(n \cdot k \cdot N^2)$.

Step 3 constructs G, whose size is at most $\min\{k^n, N^{\rho^*}\}$, since there are k means per attribute and n attribute and the coreset cannot exceed the size of D. In practice, this number can be much smaller since we skip the data points whose weights are zero. The query (9.21) takes time $O(|\mathcal{V}|^2 \cdot |\mathcal{E}| \min\{N^{\mathsf{fhtw}} k^m, N^{\rho^*}\} \log N)$. We can explain this runtime upper bound in two ways. One approach is to compute all tuples $(x_1, \ldots, x_n)$ in the result of the feature extraction query. This takes time $O(|\mathcal{V}|^2 \cdot |\mathcal{E}| N^{\rho^*} \log N)$. We then add the functionally determined variable c_j for each variable x_j, discard the variables x_j and count the occurrences of each distinct tuple $(c_1, \ldots, c_m)$ in time proportional to the size $|D| = O(N^{\rho^*})$ of the query result D. A second approach is to compute a hypertree decomposition of the feature extraction query. This takes time $O(|\mathcal{V}| \cdot |\mathcal{E}| \cdot N^{\mathsf{fhtw}} \log N)$. For each data point g in G of size b (out of at most k^n many), we select at each bag of the decomposition the tuples agreeing with c and then compute the count over the reduced decomposition in time $O(|\mathcal{V}|^2 \cdot |\mathcal{E}| N^{\mathsf{fhtw}} \log N)$.

Step 4 clusters G in time $O((|G|kn + Nkn)it)$, where it is the number of iterations of k-means used in Step 4; we assume this to be constant with respect to the database size.

The most expensive computation is due to the one-dimensional clustering for the continuous variables and the computation of the coreset. □

9.3 Mutual Information

In this section, we consider computing the mutual information (MI) between two distinct discrete random variables X and Y. MI is a measure of their mutual dependence and quantifies the amount of information one can obtain about one random variable through observing the other random variable.

Let $p_{(X,Y)}(x, y)$ denote the joint probability mass function of the discrete random variables X and Y, and $p_X(x)$ is the marginal probability mass function of X. Then, the mutual information of two discrete random variables X and Y is defined as follows:

$$I(X;Y) = \sum_{x \in \mathsf{Dom}(X)} \sum_{y \in \mathsf{Dom}(Y)} p_{(X,Y)}(x,y) \cdot \log(\frac{p_{(X,Y)}(x,y)}{p_X(x) \cdot p_Y(y)}) \tag{9.26}$$

A high value for $I(X;Y)$ means that the random variables X and Y share a lot of information. In contrast, if X and Y are independent, i.e., $p_{(X,Y)}(x, y) = p_X(x) \cdot p_Y(y)$, they do not share any information and $I(X;Y) = 0$.

Mutual information has many applications in machine learning. We highlight three such applications:

- Mutual information is used as a criterion for feature selection. Consider the case where we learn a supervised learning model over a training dataset D with variables $X_1, \ldots, X_n$ and label Y. To decide which variables are relevant features for the model, we first compute the mutual information $I(X_i; Y)$ between each variable $(X_I)_{i \in [n]}$ and the label Y, and then choose the variables with the highest mutual information with Y as features in the desired model.

- Mutual information is also used as a cost function in learning classification trees. In such scenarios, the mutual information encodes the information gained for Y after splitting the dataset with a condition on variable X_i. Therefore, mutual information is an alternative cost function for the entropy and Gini index from Chapter 8.

- The mutual information is used in learning the structure of Bayesian networks. For instance, the Chow-Liu algorithm [40] constructs an optimal tree-shaped Bayesian network T with one node for each input variable in the set $\mathbf{X} = \{X_1, \ldots, X_n\}$. It proceeds in rounds and in each round it adds to T an edge (X_i, X_j) between the nodes X_i and X_j such that the mutual information of X_i and X_j is maximal among all pairs of variables not chosen yet.

We next consider computing mutual information over databases. Let Q denote a feature extraction query with (discrete) variables $X_1, \ldots, X_n$ that is computed over a database I with relations $R_1, \ldots, R_m$.

In this setting, we can estimate the probability mass functions for variable X_i using its active domain $\mathsf{Dom}(X_i)$ in D. This amounts to computing the frequencies for each value $x_i \in \mathsf{Dom}(X_i)$. Similar, we can estimate the joint probability mass functions for two random variables X_i and X_j by computing the frequency of pairs $(x_i, x_j) \in \mathsf{Dom}(X_i) \times \mathsf{Dom}(X_j)$. We capture the frequencies for the mutual information for X_i and X_j using the following

count queries that group by any subset of $\{X, Y\}$:

$$Q(\texttt{SUM(1)}) \leftarrow R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}) \tag{9.27}$$

$$Q_i(X_i, \texttt{SUM(1)}) \leftarrow R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}) \tag{9.28}$$

$$Q_j(X_j, \texttt{SUM(1)}) \leftarrow R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}) \tag{9.29}$$

$$Q_{(ij)}(X_i, X_j, \texttt{SUM(1)}) \leftarrow R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}) \tag{9.30}$$

The mutual information of X_i and X_j is then given by the following query with a 4-ary aggregate function f over the aggregates of the queries defined above:

$$\mathsf{MI}(f(\alpha, \beta, \gamma, \delta)) \leftarrow Q(\alpha), Q_i(X_i, \beta), Q_j(X_j, \gamma), Q_{(ij)}(X_i, X_j, \delta)$$
$$f(\alpha, \beta, \gamma, \delta) \leftarrow \frac{\delta}{\alpha} \cdot \log\left(\frac{\alpha \cdot \delta}{\beta \cdot \gamma}\right)$$

Complexity analysis

Based on queries (9.27)-(9.30), we can give following runtime bound for computing mutual information. The applications mentioned above compute the mutual information for many pairs of random variables, each of which can be computed with the same runtime.

Theorem 9.7. *Let Q be a feature extraction query with discrete variables $X_1, \ldots, X_n$ that is computed over a database instance I, where all relations have size at most N. Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ denote the query hypergraph of Q.*

We can compute the mutual information for any two discrete variables X_i and X_j over Q in time:

$$O\left(|\mathcal{V}|^2 \cdot |\mathcal{E}| \cdot N^{\mathsf{fhtw}(\mathcal{H})+1} \cdot \log N\right). \tag{9.31}$$

Proof. We have shown that the core computation for mutual information between two discrete variables over a feature extraction query Q can be expressed as four group-by aggregate queries of the form (4.1). Each of these queries computes one count aggregate ($s = 1$), and has at most two free variables. The runtime bound for computing the mutual information between two discrete variables over a feature extraction query Q then follows directly from Theorem 5.15 and Proposition 5.13, which state that we can compute a query φ of the form (4.1) in time: $O\left(|\mathcal{V}|^2 \cdot |\mathcal{E}| \cdot s \cdot (N^{\mathsf{fhtw}(\mathcal{H})+1} + |\varphi|) \cdot \log N\right)$. For the queries that define the data-intensive computation for mutual information, we know that $s = 1$, and $|\varphi| = O(N^2)$ is always upper bounded by $N^{\mathsf{fhtw}(\mathcal{H})+1}$. □

9.4 Data Cubes

Data cubes [59] are popular in decision support systems and data warehousing scenarios. A data cube is a representation of multi-dimensional data, where each dimension is a variable

in the dataset and the values represent aggregations over predefined measures of interest. The data cube provides information on the measure of interest for each subset of the dimensions. In practice, data cubes are represented as tables in 1NF using a special ALL value standing for a set of values.

Example 9.8. We consider a typical retail forecasting scenario where the data cube is used to analyze the sales for each item, store, and date in a year. The data cube then has three dimensions that represent stores, items, and dates, and the measure of interest is the number of units sold.

A data cube readily provides the information on the measure of interest for any subset of the dimensions. For instance, it represents the number of units sold for each store, item and date, but also the number of units sold for each store at a given date, where the measure aggregates over all items. In the latter, the corresponding tuples that define the key of the data cube are triples (s, d, ALL), where $s \in \mathsf{Dom}(\text{store})$, $d \in \mathsf{Dom}(\text{date})$, and ALL indicates that the measure aggregates over all items in $\mathsf{Dom}(\text{item})$. □

Let Q denote a feature extraction query with variables $\boldsymbol{X} = \{X_1, \ldots, X_n\}$ that is computed over a database I with relations $R_1, \ldots, R_m$. For a set $S_k \subseteq \mathbf{X}$ of k variables or dimensions, a k-dimensional data cube is a shorthand for the union of 2^k cube aggregates with v measure variables $V_1, \ldots, V_v \in \mathbf{X}$, where each variable V_i uses the same aggregation function α for all cube aggregates. We define one cube aggregate for each of the 2^k possible subsets of S_k:

$$\forall F_i \subseteq S_k : \ \mathsf{Cube}_i(F_i, \texttt{SUM(}\alpha_1\texttt{)}, \ldots, \texttt{SUM(}\alpha_v\texttt{)}) \leftarrow R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}) \tag{9.32}$$

Note that each cube aggregate denotes a group-by aggregate queries of the form (4.1).

Complexity analysis

Based on the cube aggregate query (9.32), we can now define the complexity of computing a data cube with dimensions S_k over feature extraction queries.

Theorem 9.9. *Let Q be a feature extraction query with variables $X_1, \ldots, X_n$ that is computed over a database instance I where all relations have size at most N. Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ denote the query hypergraph of Q.*

We can compute a k-dimensional data cube with dimensions $S_k \subseteq \mathcal{V}$ and v measure variables over Q in time:

$$O\left(|\mathcal{V}|^2 \cdot |\mathcal{E}| \cdot v \cdot \left(\sum_{F_i \subseteq S_k} \cdot N^{\mathsf{fhtw}_{F_i}(\mathcal{H})} + N^{\rho^*_{\mathcal{H}}(F_i)}\right) \cdot \log N\right) \tag{9.33}$$

Proof. We have shown that the k-dimensional data cube with dimensions S_k requires the computation of 2^k group-by aggregate queries of the form (4.1).

The runtime for computing the entire data cube directly follows from Theorem (5.15), which states that each query φ of the form (4.1) with v aggregates and group-by variables F can be computed in time $O(|\mathcal{V}|^2 \cdot |\mathcal{E}| \cdot v \cdot N^{\mathsf{fhtw}_F(\mathcal{H})} + |\varphi|) \cdot \log N)$. In addition, following the AGM bound [19] states that $|\varphi| \leq O(N^{\rho^*_{\mathcal{H}}(F)})$, assuming the result is provided as listing representation (cf. Section 5.3). □

Chapter 10

Related Work

In this chapter, we overview related work, which consists of three parts: (1) we summarize the current approaches for learning models over multi-relational databases; (2) we overview techniques used by the database theory community to optimize machine learning workload; (3) we discuss two techniques used in machine learning that exploit the structure of the data and problem: sparse data representations, and coresets; and (4) we overview related work for aggregate queries with inequalities.

In addition, Chapters 4 and 5 already introduced related work on state-of-the-art query evaluation techniques and width measure of query hypergraphs. Chapter 16 overviews systems techniques that are commonly employed in machine learning and databases.

Approaches to Machine Learning over Databases

Our work follows closely Chaudhuri's manifesto on SQL-aware data mining systems from two decades ago [32] in two key aspects. First, the goal of our work is not to invent new machine learning models or data analysis techniques, but identify common data-centric steps across a broad class of learning algorithms and investigate their theoretical and systems challenges. We show that such steps can be encoded as SQL group-by aggregate queries, which are amenable to shared batch computation. Second, our approach performs data analysis not only over materialized relations but more importantly over feature extraction queries, whose results need not be materialized. This enables the interaction between the aggregates encoding the data-centric steps and the underlying queries (this is called ad-hoc mining in Chaudhuri's terminology).

A reevaluation of Chaudhuri's manifesto in today's context brings forth two important technical changes. The first game-changer is represented by the recent development on query processing. This includes a new breed of worst-case optimal join algorithms, which support listing representations [96, 133] and factorized representations [102] of query results, and extensions to aggregate computation [23, 10, 101]. These algorithms exploit developments on (fractional) hypertree decompositions of relational queries [58, 60, 84]. These

algorithms overshadow the traditional query plans in both asymptotic complexity [97] and practical performance [23, 98]. The second change is in the workload. Whereas SQL-aware data mining systems were mostly concerned with association rules, decision trees, and clustering, current workloads feature a broader spectrum of increasingly more sophisticated machine learning (ML) models, including polynomial regression models, factorization machines, principal component analysis, generalized linear models, generalized low-rank models, sum-product networks, and convolutional networks. There is also a more profound orthogonal change: There is more data readily available in all aspects of our society and there is more appetite in industry to monetize it by turning it into knowledge.

The current landscape for analytics solutions over multi-relational data can be categorized depending on the degree of integration of the data system, which hosts the data and supports data access via queries, with the ML library of models and learning algorithms.

No integration of databases and machine learning

By far the most common solutions provide no integration of the two systems, which are distinct tools on the technology stack: The data system first computes the result of a feature extraction query and exports it as one table commonly in CSV or binary format. The statistical package then imports the training dataset into its own format and learns the desired model. The first step is typically performed in open-source data systems such as MySQL/PostgreSQL or SparkSQL [139]. The second step commonly uses ML libraries such as R [112], Python StatsModels [131], scikit-learn [106], MLpack [41], TensorFlow [1], SystemML [68, 25], MLLib [88], and XGBoost [37]. As an alternative to data-management systems, one can also compute the feature extraction query in query processing libraries, such as Python Pandas [86] or R dplyr [136]. The advantage is that these libraries share the same environment with the statistical package, but they are in-memory libraries without a storage layer. Therefore, they require yet another system to store the data.

The advantage of this approach is that the two systems can be developed independently, with virtually any ML model readily available for use. Two disadvantages of such common solutions are the expensive data export/import at the interface between the two systems and the materialization of the training dataset as a result of a feature extraction query over multi-relational data. The feature extraction query is computed inside the data system, its result exported and imported into the data format of the ML system, where the model is learned. Furthermore, the materialized training dataset may be much larger than the input data (cf. Table 18.1). This is exacerbated by the stark asymmetry between the two systems: Whereas data systems tend to scale to large datasets, this is not the case for ML libraries. Yet, such solutions expect by design that the ML libraries work on even larger inputs than the data systems! A further disadvantage is that these solutions inherit the limitations of both underlying systems. For instance, the R data frame can host at most 2^{31} values, which makes it impossible to learn models over large datasets. Database systems

can only handle up to a few thousand columns per relation, which can be smaller than the number of features of the model.

Loose integration of databases and machine learning

The second class of systems features a loose integration, even though they remain structure-agnostic: The ML code migrates inside the space of the data system process, with each ML task being implemented by a distinct user-defined aggregate function (UDAF). Prime examples of this class are MADlib [67] and GLADE PF-OLA [109]. MADlib casts analytics as UDAFs that can be used in SQL queries and executed inside PostgreSQL. GLADE PF-OLA casts analytics as a special form of UDAFs called Generalized Linear Aggregates that can be executed using the GLADE distributed engine [38]. These UDAFs remain black boxes for the underlying query engine, which has to compute the feature extraction query and delegate the UDAF computation on top of the query result to the MADLib's and GLADE PF-OLA's specialized code. The advantage of this approach is that the expensive export/import step is avoided. The disadvantage is that each ML task has to be migrated inside the data system space, which comes with design and implementation overhead.

A variation of the second approach provides a *unified programming architecture*, one framework for many machine learning tasks instead of one distinct UDAF per task, with possible code reuse across UDAFs. Prime example of this approach is Bismark [50], a system that supports incremental (stochastic) gradient descent for convex programming. Its drawback is that its code may be less efficient than the specialized UDAFs. Code reuse across various models and optimization problems may however speed up the development of new functionalities such as new models and optimization algorithms.

Tight integration of databases and machine learning

The third class of systems features a tight integration and is structure-aware: There is one execution strategy for both the feature extraction query and the subsequent learning task, with components of the latter possibly pushed past the joins in the former. This plan works directly on the input data and computes sufficient statistics of much smaller size than of the training dataset. Our system LMFAO presented in Chapter 11 is a prime example of this approach. It builds on F [119] and AC/DC [6]. F supports linear regression models. AC/DC generalizes F to polynomial regression models and factorization machines, as well as the efficient support for categorical variables. A key aspect that sets apart F, AC/DC, and LMFAO from all other efforts is the use of execution plans for the mixed workload of queries and learning whose complexity may be asymptotically lower even than that of the materialization step. In particular, this line of work shows that all machine learning approaches that require as input the materialization of the result of the feature extraction query may be asymptotically suboptimal.

Figure 1.1 sums up the difference between the first two classes that fall under structure-agnostic learning and the third class that broadly represents structure-aware learning. The inspiration for our work lies with factorized computation of aggregates over joins [23, 10], which avoids the materialization of joins, and with the LogicBlox system [89, 16], which has a unified system architecture and declarative programming language for hybrid database and optimization workloads.

Further examples in this category are: Orion [79] and Hamlet [80], which support generalized linear models and Naïve Bayes classification; recent efforts on scaling linear algebra using existing distributed database systems [83]; the declarative language BUDS [52], whose compiler can perform deep optimizations of the user's program; and Morpheus [36]. Morpheus factorizes the computation of linear algebra operators summation, matrix-multiplication, pseudo-inverse, and element-wise operations over training datasets defined by key-foreign key star or chain joins. It represents the training dataset as a normalized matrix, which is a triple of the fact table, a list of dimension tables, and a list of indicator matrices that encode the join between the fact table and each dimension table. Morpheus provides operator rewritings that exploit the relational structure by pushing computation past joins to the smaller dimension tables. Initial implementations are built on top of the R and Python numpy linear algebra packages. Morpheus only supports key-foreign key star or chain joins and models that are expressible in linear algebra. In contrast, LMFAO supports arbitrary joins and rich aggregates that can capture computation needed by a large heterogeneous set of models beyond those expressible in linear algebra, including, e.g., decision trees.

Machine Learning through the Lens of Database Theory

It has been recently acknowledged that database theory can effectively contribute to the arms race for in-database analytics [4]. Recent works highlight the potential of applying key database theory tools to this growing research of practical interest, e.g., the formal relational framework for classifier engineering [76] and in-database factorized learning of regression models with low data complexity [119, 7].

There are three lines of prior work closest to ours.

One line of work investigates the ability to express parts of analytical tasks within query languages. Very recent works investigate query languages for matrices [29] and a relational framework for classifier engineering [76]. They follow works on query languages with data mining capabilities [27, 103], also called descriptive or backward-looking analytics, capabilities, and on in-database data mining solutions, e.g., frequent itemsets [107] and association rule mining [12]. More recent work investigated how to (partly) express predictive (forward-looking) analytics, such as learning regression models and Naïve Bayes classification, together with the feature extraction query as a single optimized query with joins and sum-product aggregates [79, 119, 7, 118, 5]. Our rewriting of ML code into aggre-

gates falls into this line of work as well. The additional fixpoint computation needed on top of the aggregate computation for convergence of the model parameters, which is intrinsic to gradient descent approaches, can be expressed using recursive queries [3].

A second line of work exploits join dependencies for efficient in-database analytics. Join dependencies form the basis of the theory of (generalized) hypertree decompositions [58] and factorized databases [102], with applications such as inference in probabilistic graphical models, CSP, SAT, and databases. In databases, they have been originally used as a tractability yardstick for Boolean conjunctive queries [58] and more recently for the computation and result representation of queries with free variables [102], with group-by aggregates [23, 10], and with order-by clauses [23]. Our approach builds on earlier work that exploits join dependencies for learning linear regression models with continuous features [119]. Factorization machines [116] represent a regression model used for real-world analytics and that we investigate in this paper. In contrast to polynomial regression models, factorization machines factorize the space of model parameters to better capture data correlations. We further this idea by also factorizing the training dataset, which relies on join dependencies present in the data.

A third line of prior work uses functional dependencies (FDs) to simplify the computation of the model. AC/DC uses FDs to reparameterize polynomial regression models and factorization machines, and learns a simpler, equivalent model instead [7, 6]. Hamlet avoids key-foreign key joins that satisfies certain degree constraints to reduce the number of features in Naïve Bayes classification and logistic regression models [80].

Structure exploited by Machine Learning Systems

We overview sparse data representations and coresets, which are two techniques commonly used in machine learning to exploit the structure of the data and/or problem.

State-of-the-art machine learning systems use a sparse representation of the input data to avoid redundancy introduced by one-hot encoding [115, 48]. In our setting, however, such systems require an additional data transformation step after the result of the feature extraction query is exported. This additional step is time consuming and makes the use of such systems inefficient in many practical applications. In statistics and machine learning, there is a rich literature on learning with sparse and/or multilinear structures [66]. Such methods complement our framework and it would be of interest to leverage and adapt them to our setting.

Recent developments in machine learning exploit the structure of the model as well as the underlying data matrix D to construct a coreset of D. A coreset of D is a small set of points that provides a good summarization of the original dataset D. The aim is to accelerate the learning of the model with good approximation guarantees. Coresets have been applied to a variety of machine learning models, including k-means [63, 21], support

vector machines [64], bayesian inference [30], and regression models [20]. These coresets, however, require that the underlying data matrix is materialized, and thus fail to exploit the relational structure for models learned over databases. The Rk-means algorithm from Section 9.2.2 performs k-means clustering over coresets, while also exploiting the relational structure in the underlying data. This is possible because the *grid coreset* constructed by Rk-means does not require the materialization of the feature extraction query.

Aggregate Queries with Additive Inequalities

We consider related work for aggregate queries with additive inequalities. Chapter 5 already contrasts our new width notions with fhtw and subw, and overviews related work on geometric data structures.

A seminal work considers the containment and minimization problem for queries with inequalities [78]. There is a bulk of work on queries with *dis*equalities (not-equal), e.g., [9] and references therein, which are at times referred to as inequalities.

In the following, we revisit two prior results on the evaluation of queries with inequalities through the lens of aggregate queries with additive inequalities lenses: Core XPath queries over XML documents and inequality joins over tuple-independent probabilistic databases [99]. Our main observation is that their linearithmic complexity is due to the same structural property behind relaxed tree decompositions: Such queries admit trivially a relaxed tree decomposition, where each bag corresponds to one relation in the query and the inequality edges, i.e., the inequality joins, are covered by neighboring bags.

Core XPath Queries

We consider the problem of evaluating Core XPath queries over XML documents. An XML document is represented as a rooted tree whose nodes follow the *document order*. Core XPath queries define traversals of such trees using two constructs: (1) a *context node* that is the starting point of the traversal; and (2) a tree of *location steps* with one distinguished branch that selects nodes and all other branches conditioning this selection. Given a context node v, a location step selects a set of the nodes in the tree that are accessible from v via the step's *axis*. This set of nodes provides the context nodes for the next step, which is evaluated for each such node in turn. The result of the location step is the set of nodes accessible from any of its input context nodes, sorted in document order.

The *preorder rank* $pre(v)$ of a node v is the index of v in the list of all nodes in the tree that are visited in the (depth-first, left-to-right) preorder traversal of the tree; this order is the document order. Similarly, the *postorder rank* $post(v)$ of v is its index in the list of all nodes in the tree that are visited in the (depth-first, left-to-right) postorder tree traversal. We can use the pre/post-order ranks of nodes to define the main axes `descendant`, `ancestor`, `following`, and `preceding` [61]. Given two nodes v and v' in the tree, the four

axes are defined using the pre/post two-dimensional plane:

- v' is a descendant of v or equivalently v is an ancestor of v'

$$\text{iff } pre(v) < pre(v') \wedge post(v') < post(v)$$

- v' follows v or equivalently v precedes v'

$$\text{iff } pre(v) < pre(v') \wedge post(v) < post(v')$$

The remaining axes `parent`, `child`, `following-sibling`, and `preceding-sibling` are restrictions of the four main axes, where we also use the parent information *par* for each node:

- v' is a child of v or equivalently v is a parent of v'

$$\text{iff } v = par(v')$$

- v' is a following sibling of v or equivalently v is a preceding sibling of v'

$$\text{iff } pre(v) < pre(v') \wedge post(v) < post(v') \wedge par(v) = par(v')$$

We follow the standard approach to reformulate XPath evaluation in the relational domain [61]. We represent the document by a factor $\boldsymbol{G}$ in the Boolean semiring with schema $(pre, post, par, tag)$. For each node in the tree there is one tuple in $\boldsymbol{G}$ with *pre* and *post* ranks, label *tag*, and preorder rank *par* of the parent node. A query with n location steps is mapped to an aggregate query Q with additive inequalities that is a join of $n+1$ copies of $\boldsymbol{G}$ where the join conditions are the inequalities encoding the axes of the n steps. The first copy $\boldsymbol{G}_0$ is for the initial context node(s). The axis of the i-th step is translated into the conjunction of inequalities between pre/post rank variables of the copies $\boldsymbol{G}_{i-1}$ and $\boldsymbol{G}_i$. The query Q has one free variable: This is the preorder rank variable from the copy of $\boldsymbol{G}$ corresponding to the location step that selects the result nodes.

Example 10.1. The Core XPath query

$$v/\text{descendant} :: a[\text{descendant} :: c]/\text{following} :: b$$

selects all b-labeled nodes following a-labeled nodes that are descendants of the given context node v and that have at least one c-labeled descendant node. The steps in the above textual representation of the query are separated by /. The brackets [ ] delimit a condition on the selection of the a-labeled nodes. We can reformulate this a conjunctive query as query of

the form (4.1) as follows:

$$
\begin{aligned}
Q(pre_b) \leftarrow & \boldsymbol{G}_v(pre_v, post_v, p_v, tag_v), \boldsymbol{G}_a(pre_a, post_a, p_a, \text{'a'}), \\
& \boldsymbol{G}_c(pre_c, post_c, p_c, \text{'c'}), \boldsymbol{G}_b(pre_b, post_b, p_b, \text{'b'}), \\
& pre_v < pre_a, post_a < post_v, \quad \text{// a is descendant of v} \\
& pre_a < pre_c, post_c < post_a, \quad \text{// c is descendant of a} \\
& pre_a < pre_b, post_a < post_b \quad \text{// b is following a}
\end{aligned}
$$

□

The hypergraph of a relational encoding of a Core XPath query has one relation hyperedge for each copy of the document factor and one inequality edge for each pair of inequalities over two of these copies. Any two relation hyperedges may only have one node, i.e., query variable, in common to express the parent/child or sibling relationship between their corresponding steps. This hypergraph admits a trivial relaxed tree decomposition, which mirrors the tree structure of the query. In particular, there is one bag of the decomposition consisting of the variables of each copy of the document factor. Each inequality edge represents a pair of inequalities over variables of two neighboring bags. The running intersection property holds since the equalities are by construction only over variables from neighboring bags.

It is known that the time complexity of answering a Core XPath query Q with n location steps over an XML document $\boldsymbol{G}$ is $O(n \cdot |\boldsymbol{G}|)$ (Theorem 8.5 [57]; it assumes the document factor sorted). We can show a linearithmic time complexity result using the aggregate query reformulation of Core XPath queries and the trivial relaxed tree decomposition.

Proposition 10.2. *For any Core XPath query Q with n location steps and XML document $\boldsymbol{G}$, the query answer can be computed in time $O(n \cdot |\boldsymbol{G}| \cdot \log |\boldsymbol{G}|)$.*

Proof. Let φ be the aggregate query reformulation of Q and F the factor representing the XML document $\boldsymbol{G}$. There is a one-to-one correspondence between the trivial relaxed tree decomposition and the XPath query, with one bag per location step. Let n be the number of location steps in Q, or equivalently the number of bags in the tree decomposition. We consider this trivial tree decomposition and choose its root as the bag corresponding to the location step that selects the answer node set. Our evaluation algorithm proceeds in a bottom-up left-to-right traversal of the tree decomposition and eliminates one bag at a time. This bag elimination is a variant

We index the bags and their corresponding factors in this traversal order. The first factor to eliminate is then denoted by F_1 while the last factor, which corresponds to the location step selecting the answer node set, is denoted by F_n.

We initially create factors S_j that are copies of factors F_j corresponding to leaf bags in the tree. Consider now two factors S_j and F_i corresponding to a leaf bag and respectively to its parent bag. Let $\phi_{i,j}$ be the conjunction of inequalities defining the axis relationship

between the location steps corresponding to these bags. We then compute a new factor S_i that consists of those tuples in F_i that join with some tuples in S_j. This is expressed in the following aggregate query:

$$S_i(pre_i, post_i, p_i, t_i) \leftarrow F_i(pre_i, post_i, p_i, t_i), S_j(pre_j, post_j, p_j, t_j), \phi_{i,j}$$

The conjunction $\phi_{i,j}$ only has two inequalities on variables between the two bags. Computing S_i takes time $O(|F| \log |F|)$ following the algorithm from the proof of Theorem 5.18. We can sort both F_i and S_j in ascending order on the preorder column and in descending order on the postorder column. For each tuple t in F_i, the tuples in S_j that join with t form a contiguous range in S_j. To assert whether t is in S_i, it suffices to check that this range is not empty. There are n such steps and $|F| = |F_i| = |\boldsymbol{G}|$, with an overall time complexity of $O(n \cdot |\boldsymbol{G}| \log |\boldsymbol{G}|)$. □

Probabilistic Queries with Inequalities

The problem of query evaluation in probabilistic databases is #P-hard for general queries and probabilistic database formalisms [129]. Extensive prior work focused on charting the tractability frontier of this problem, with positive results for several classes of queries on so-called tuple-independent probabilistic databases. We discuss here one such class of queries with inequality joins called IQ [99].

A tuple-independent probabilistic database is a database where each tuple t is associated with a Boolean random variable $v(t)$ that is independent of the other tuples in the database. This is the database formalism of choice for studies on query tractability since inference is hard already for trivial queries on more expressive probabilistic database formalisms [129].

In the following, we use functional aggregate query (FAQ) syntax [10] to denote the queries, because FAQ factors naturally capture tuple-independent probabilistic databases: A tuple-independent probabilistic relation R is a factor that maps each tuple t in R to the probability that the associated random variable $v(t)$ is true.

We next define the class IQ of inequality queries and later show how to recover the linearithmic time complexity for their inference.

Definition 10.3 (adapted from Definitions 3.1, 3.2 [99])**.** Let a hypergraph $\mathcal{H} = (\mathcal{V} = [n], \mathcal{E}_s \cup \mathcal{E}_\ell)$, where $\mathcal{E}_s$ and $\mathcal{E}_\ell$ are disjoint, $\mathcal{E}_s$ consists of pairwise disjoint sets, $\mathcal{E}_\ell$ consists of sets $\{i, j\}$ for which there is a vector $c_{i,j} \in \{[1, -1]^{\mathrm{T}}, [-1, 1]^{\mathrm{T}}\}$, and $\forall F \in \mathcal{E}_s : |(\bigcup_{I \in \mathcal{E}_\ell} I) \cap F| \leq 1$. An IQ query has the form

$$Q() \leftarrow \bigwedge_{F \in \mathcal{E}_s} R_F(\boldsymbol{X}_F) \wedge \bigwedge_{\{i,j\} \in \mathcal{E}_\ell} [X_i, X_j]^{\mathrm{T}} \cdot c_{i,j} \leq 0 \tag{10.1}$$

where $(R_F)_{F \in \mathcal{E}_s}$ are distinct factors. □

The edges (i.e., binary hyperedges) in $\mathcal{E}_\ell$ correspond to inequalities of the query variables. These inequalities are restricted so that there is at most one node (query variable) from any hyperedge in $\mathcal{E}_s$. Inequalities on variables of the same factor are not in $\mathcal{E}_\ell$; they can be computed trivially in a pre-processing step.

The inequalities may only have the form $X_i \leq X_j$ or $X_j \leq X_i$. They induce an *inequality graph* where X_i is a parent of X_j if $X_i \leq X_j$. This graph can be minimized by removing edges corresponding to redundant inequalities implied by other inequalities [71]. Each graph node thus corresponds to precisely one factor. We categorize the IQ queries based on the structural complexity of their inequality graphs into (forests of) paths, trees, and graphs.

Example 10.4. Consider the following IQ queries:

$$Q_1() \to R(A) \wedge S(B) \wedge T(C) \wedge A \leq B \wedge B \leq C$$
$$Q_2() \to R(A) \wedge S(B) \wedge T(C) \wedge A \leq B \wedge A \leq C$$

The inequalities form a path in Q_1 and a tree in Q_2. □

The probability a query over a probabilistic database $\boldsymbol{I}$ is the probability of its *lineage* [129]. The lineage is a propositional formula over the random variables associated with the input tuples. It is equivalent to the disjunction of all possible derivations of the query answer from the input tuples.

Example 10.5. Consider the factors R, S, T, where r_i, s_j, t_k denote the variables associated with the tuples in these factors and for a random variable a, p_a denotes the probability that $a =$ true:

R	A	
r_1	1	p_{r_1}
r_2	2	p_{r_2}
r_3	3	p_{r_3}

S	B	
s_1	2	p_{s_1}
s_2	3	p_{s_2}
s_3	4	p_{s_3}

T	C	
t_1	3	p_{t_1}
t_2	4	p_{t_2}
t_3	5	p_{t_3}

The lineage of Q_1 and Q_2 over these factors is:

$$\underbrace{\begin{aligned} &r_1[s_1(t_1+t_2+t_3)+s_2(t_2+t_3)+s_3t_3]+\\ &r_2[\qquad\qquad s_2(t_2+t_3)+s_3t_3]+\\ &r_3[\qquad\qquad\qquad\quad s_3t_3]+ \end{aligned}}_{\text{lineage of } Q_1} \qquad \underbrace{\begin{aligned} &r_1(s_1+s_2+s_3)(t_1+t_2+t_3)+\\ &r_2(\quad s_2+s_3)(t_1+t_2+t_3)+\\ &r_3(\quad\quad s_3)(\quad t_2+t_3) \end{aligned}}_{\text{lineage of } Q_2}$$

□

Prior work (Theorem 4.7 [99]) showed that the probability of an IQ query Q with an inequality tree with k nodes over a tuple-independent probabilistic database of size N can be computed in time $O(2^k \cdot N \log N)$ using a construction of the query lineage in an Ordered Binary Decision Diagram (OBDD). We show next that a variant of the algorithm in the proof

of Lemma 5.17, adapted from counting to *weighted counting*, i.e., probability computation, can compute the probability in time $O(N \log N)$, thus shaving off an exponential factor in the number of inequalities.

We first explain this result using two examples, which draw on a crucial observation made in prior work [99]: The lineage of IQ queries has a chain structure: For each factor, there is an order on its random variables that defines a chain of logical implications between their cofactors in the lineage: the cofactor of the first variable implies the cofactor of the second variable, which implies the cofactor of the third variable, and so on.

Example 10.6. We continue Example 10.5. The lineage of Q_1 and Q_2 is arranged so that the chain structure becomes apparent. This structure allows for an equivalent rewriting of the lineage [99], as shown next for the lineage ϕ_{r_1} of Q_1 (for a random variable a, $\overline{a}$ denotes its negation):

$$\begin{aligned}
\phi_{r_i} &= r_i \phi_{s_i} + \overline{r_i} \phi_{r_{i+1}}, \forall i \in [3]; \qquad \phi_{r_4} = \text{false} \\
\phi_{s_j} &= s_j \phi_{t_j} + \overline{s_j} \phi_{s_{j+1}}, \forall j \in [3]; \qquad \phi_{s_4} = \text{false} \\
\phi_{t_k} &= t_k \quad + \overline{t_k} \phi_{t_{k+1}}, \forall k \in [3]; \qquad \phi_{t_4} = \text{false}
\end{aligned}$$

In disjunctive normal form, the lineage of Q_1 may have size cubic in the size of the database. The factorization of the lineage in Example 10.5 lowers the size to quadratic. The above rewriting further reduces the size to linear. The rewritten form can be read directly from the input factors following the structure of the inequality tree.

Since the above expressions are sums of two mutually exclusive formulas, their probabilities are the sums of the probabilities of their respective two formulas. Their probabilities can be computed in one bottom-up right-to-left pass: First for ϕ_{t_k} in decreasing order of k, then for ϕ_{s_j} in decreasing order of j, and finally for ϕ_{r_i} in decreasing order of i. We extend the probability function p from input random variables to formulas over these variables. The probability of Q_1's lineage, which is also the probability of Q_1, is ($\forall i, j, k \in [3]$):

$$\begin{aligned}
p(\phi_{r_i}) &= p(r_i) \cdot p(\phi_{s_i}) + [1 - p(r_i)] \cdot p(\phi_{r_{i+1}}) \\
p(\phi_{s_j}) &= p(s_j) \cdot p(\phi_{t_j}) + [1 - p(s_j)] \cdot p(\phi_{s_{j+1}}) \\
p(\phi_{t_k}) &= p(t_k) \qquad\quad + [1 - p(t_k)] \cdot p(\phi_{t_{k+1}})
\end{aligned}$$

Since there are no variables r_4, s_4, and t_4, we use $p(\phi_{r_4}) = p(\phi_{s_4}) = p(\phi_{t_4}) = 0$. This computation corresponds to a decomposition of ϕ_{r_1} that can be captured by a linear-size OBDD [99].

The probability of the lineage ψ_{r_1} of Q_2 is computed similarly ($\forall i, j, k \in [3]$):

$$
\begin{aligned}
p(\psi_{r_i}) &= p(r_i) \cdot p(\psi_{s_i}) \cdot p(\psi_{t_i}) + [1 - p(r_i)] \cdot p(\psi_{r_{i+1}}) \\
p(\psi_{s_j}) &= p(s_j) + [1 - p(s_j)] \cdot p(\psi_{s_{j+1}}) \\
p(\psi_{t_k}) &= p(t_k) + [1 - p(t_k)] \cdot p(\psi_{t_{k+1}})
\end{aligned}
$$

This computation would correspond to a decomposition of ψ_{r_1} that can be captured by an OBDD with several nodes for a random variable from S and T; in general, such an OBDD would have a size linear in N but with an additional exponential factor in the size of the inequality tree due to the inability to represent succinctly the products of lineage over T and of lineage over S [99]. (OBDDs with AND nodes can capture such products without this exponential factor, though in this paper we do not use them.) □

Proposition 10.7. *Given a tuple-independent probabilistic database $\boldsymbol{I}$ of size N and an IQ query Q with a forest of inequality trees, we can compute the probability of Q over $\boldsymbol{I}$ in time $O(N \log N)$.*

Proof. We next present the inference algorithm for a given IQ query Q with an inequality tree. It uses a minor variant of the algorithm from the proof of Lemma 5.17 to compute a aggregate query with additive inequalities over two factors.

We first reduce the input database $\boldsymbol{I}$ to a simplified database of unary and nullary factors that is constructed by aggregating away all query variables that do not contribute to inequalities.

Let us partition $\mathcal{E}_s$ into the hyperedges $\mathcal{E}_1$ that contain query variables involved in inequalities and all other hyperedges $\mathcal{E}_2$.

We reduce each factor $(R_F)_{F \in \mathcal{E}_1}$ with a query variable X_i occurring in inequalities to a unary factor $S_{\{i\}}$ by aggregating away all other query variables. For an X_i-value x_i, $S_{\{i\}}(x_i)$ gives the probability of the disjunction of the independent random variables associated with the tuples in R_F that have the X_i-value x_i:

$$
S_{\{i\}}(x_i) = 1 - \prod_{\boldsymbol{x} \in \mathsf{Dom}(\boldsymbol{X}_F - \{i\})} \big(1 - R_F(\boldsymbol{x}_F)\big)
$$

We also reduce all factors $(R_F)_{F \in \mathcal{E}_2}$ with no query variable occurring in inequalities to one nullary factor $S_\emptyset$ by aggregating away all query variables. $S_\emptyset()$ gives the probability of the conjunction of all factors without query variables in inequalities:

$$
S_\emptyset() = \prod_{F \in \mathcal{E}_2 l} \big[1 - \prod_{\boldsymbol{x} \in \mathsf{Dom}(\boldsymbol{X}_F)} \big(1 - R_F(\boldsymbol{x}_F)\big)\big]
$$

This simplification reduces the set $\mathcal{E}_s$ of hyperedges to a new set $\mathcal{E}_u$ of unary edges, one per query variable in the inequalities, and one nullary edge: $\mathcal{E}_u = \{\emptyset\} \cup \bigcup_{\{i,j\} \in \mathcal{E}_\ell} \{\{i\}, \{j\}\}$.

The simplification does not affect the inference problem: The probability of Q is the same as the probability of the query Q' over $\mathcal{E}_u \cup \mathcal{E}_\ell$:

$$Q'() \leftarrow \bigwedge_{F \in \mathcal{E}_u} S_F(\boldsymbol{X}_F) \wedge \bigwedge_{\{i,j\} \in \mathcal{E}_\ell} [X_i, X_j]^{\mathrm{T}} \cdot c_{i,j} \leq 0 \tag{10.2}$$

The hypergraph of Q' trivially admits the relaxed tree decomposition whose structure is that of the inequality tree of Q' (and of Q): The relation edges are $\mathcal{E}_u$ and the inequality edges are $\mathcal{E}_\ell$.

The inference algorithm traverses the inequality tree bottom-up and eliminates one level of query variables at a time. For a variable X_p with children $X_{c_1}, \ldots, X_{c_k}$, it computes recursively the factor

$$Q_p(x_p) = S_p(x_p) \cdot \prod_{i \in [k]} S_{c_i}(\mathrm{lub}_i(x_p)) + (1 - S_p(x_p)) \cdot Q_p(\mathrm{lsub}_p(x_p))$$

We use $\mathrm{lub}_i(x_p)$ to find the value in S_{c_i} that is the least upper bound of x_p and $\mathrm{lsub}_p(x_p)$ to find the value in Q_p that is the least strict upper bound of x_p, i.e., the next value in ascending order. The definition of Q_p is recursive: It first computes the probability for x_p and then for its previous values. In case X_p has no children, i.e., $k = 0$ the product over S_{c_i} is one.

The probability of Q is then the product of $S_\emptyset$ and the probability of the first tuple in the factor of the root variable. If Q has a forest of inequality trees, then the subqueries for the trees would be disconnected and thus correspond to independent random variables. The probability of Q is then the product of the probabilities of the independent subqueries. □

The case of inequality graphs can be reduced to that of inequality trees by variable elimination. The elimination of a variable X_i repeatedly replaces it in the query by a value from its domain. The inequality graph of this residual query has no node for X_i and none of its edges. By removing k variables to obtain an inequality tree, the complexity of computing the query probability increases by at most the product of the sizes of the factors having these k variables.

Part II

The LMFAO System for Structure-Aware Machine Learning

Chapter 11

LMFAO: Layered Multiple Functional Aggregate Optimization

This part introduces LMFAO (Layered Multiple Functional Aggregate Optimization), an in-memory execution engine for batches of aggregate queries. LMFAO currently supports queries of the form (4.1) without additive inequalities, which are computed over the input database with relations $R_1, \ldots, R_m$:

$$Q(\omega_Q, \texttt{SUM}(\alpha)) \leftarrow R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m})$$

where ω_Q denotes the group-by (or free) variables of Q, and α is a sum of products of user-defined aggregate functions (UDAF). The efficient support of additive inequalities is left for future work.

The primary motivation for LMFAO stems from the observation in Part I that for a variety of analytics over databases, their data-intensive tasks can be decomposed into group-by aggregates over the join of the input database relations. The second follows from the observation that current database management systems fail to compute large batches of queries efficiently, as they were not designed for this workload (cf. Chapter 19).

To compute batches of aggregates efficiently, LMFAO employs several layers of optimization techniques. The optimizations are either novel or adaptations of known concepts to our specific workload, and systematically exploit factorization, sharing of computation, parallelism, and code specialization.

In the following sections, we first overview the optimization layers of LMFAO, and then expand on key design choices for each of the layers. We also provide additional information on how LMFAO captures various machine learning models that are presented in Part I.

Chapter Outline

The outline for this chapter is as follows:

- Chapter 11.1 provides an overview of the optimization layers of LMFAO.
- Chapters 12, 13, and 14 expand on the key design choices for each layer and optimization step behind LMFAO.
- Chapter 15 presents how LMFAO computes and optimizes linear regression models, decision trees, and k-means clustering.

11.1 System Overview: The Layers of LMFAO

In this chapter, we provide an overview of the layered architecture of LMFAO that is presented in Figure 11.1. There are three main optimization layers, each of which is composed of several optimization steps. We first introduce the three layers and then highlight the individual components of these layers.

LMFAO takes as input the batch of aggregate queries, the database schema, and cardinality constraints, such as sizes of relations and variable domains. The View Generation layer then decomposes each query in the batch into views over a join tree. The optimizations in this layer are concerned with logical transformations of view expressions.

The Multi-Output Optimization layer constructs groups of views, where each view group can be computed over the join of a single relation and incoming views. The layer then constructs an optimized evaluation plan for each view group, which computes all views in the group in one pass over the underlying join. Since this plan outputs the results for several views, we call it the *multi-output execution plan*. We perform several optimizations on this execution plan, which factorize the computation of the aggregates, and share the computation across views in the group. The optimizations at this layer cannot be expressed at the syntactic level of the query language.

Finally, the Code Generation layer compiles the multi-output evaluation plan from the previous layer into efficient and specialized C++ code. The components of this layer perform several low-level code optimizations, which include optimizing cache locality, inlining function calls, and parallelizing the computation. This layer is inspired by recent work on compilation for query evaluation [95, 124, 123].

We next highlight the optimization steps performed by each of the three layers.

The Join Tree step takes as input the batch of aggregates queries, the database schema, and cardinality constraints and produces one join tree that is used to compute all aggregates. This step uses known state-of-the-art techniques for acyclic queries [3]. If the query is not acyclic, we first construct a hypertree decomposition of the underlying natural join whose width matches the fractional hypertree width of the join enhanced with cardinality constraints [84], and then materialize its bags (cycles) using a worst-case optimal join algorithm [96, 133]. The outcome is a tree where each node is a (input or computed) relation.

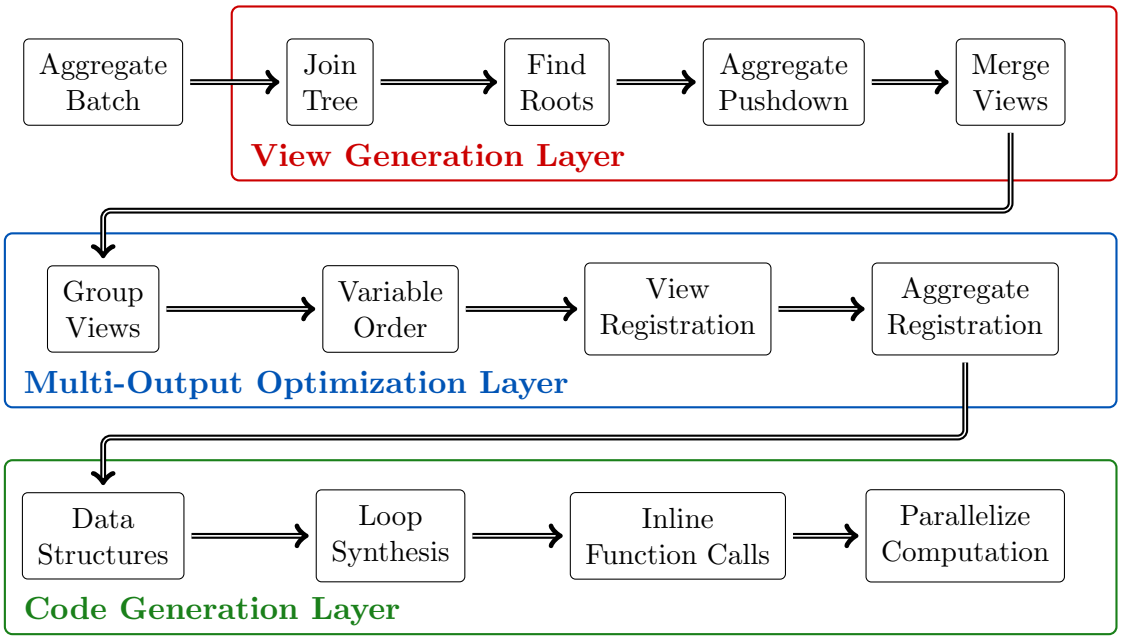


Figure 11.1: The Optimization Layers of LMFAO.

LMFAO uses a single join tree to compute all aggregates in the batch, and therefore does not match the complexity analysis for the applications in Part I. To achieve the lowest known complexity for processing aggregates with different group-by variables, it is crucial to use different join trees, where the nodes containing group-by variables form a connected subtree. We found this highly impractical as it requires too many recomputations of (parts of) the join and precludes sharing across the aggregates. To diffuse the tension between complexity and sharing, LMFAO allows for different traversals over the same join tree for different aggregates, which is facilitated by the next optimization.

The Find Roots step is novel and affects the design of all subsequent layers. By default, LMFAO computes each group-by aggregate in one bottom-up pass over the join tree, by decomposing the aggregate into views computed along each edge in the join tree. We allow for different traversals of the join tree: different aggregates are computed over the same join tree but may be rooted at different nodes. This can reduce the overall compute time for the batch as it reduces the number of views and increases the sharing of their computation. In our experiments, the use of multiple roots for the computation of aggregate batches led to $2-5\times$ speedup.

LMFAO uses *directional views* to support different traversals of the join tree: For each edge between two nodes, there may be views flowing in both directions. Directional views are similar in spirit to messages in the message passing algorithm used for inference in graphical models [105, 14]. Figure 12.1 (left) depicts directional views along a join tree.

In Aggregate Pushdown, each aggregate is decomposed into one directional view per edge in the join tree. The view at an edge going out of a node n computes the aggregate when restricted to the subtree rooted at n and is defined over the join of the views at

the incoming edges of n and of the relation at n. The directions of these views are from the leaves to the root of the aggregate. The rationale for this decomposition is twofold. First, it partially pushes the aggregates past joins (represented by edges in the tree), as in prior work on eager computation of one aggregate [137] and its generalization to factorized databases [23]. Second, it allows for sharing common views across the aggregates, which is facilitated be the next optimization step.

The Merge Views optimization merges the views generated in the previous layer. There are three types of merging possible for views along the same edge in the join tree, depending on whether they have the same: group-by variables; aggregates; and body. Views with the same direction are defined over the same subtree of the join tree. We first identify identical views constructed for different aggregates and only keep one copy. We then merge the views with the same group-by variables and body but different aggregates. We finally merge views with the same group-by variables and different bodies. This consolidation is beneficial. For instance, there are 814 aggregates to compute for learning a linear regression model over the join of five relations in our Retailer dataset (see Chapter 18). This amounts to 814 aggregates $\times$ 4 edges $= 3,256$ views, which are consolidated into 34 views that compute 1,468 aggregates.

In the Group Views step, we group views that go out of the same node, and that can be evaluated at the same time. This means that they can only be evaluated once all their incoming views used in their joins are computed. The views in a group may have different directions and group-by variables, so a view group has multiple outputs. In our example, the remaining 34 views are clustered into 7 groups.

The view group is a computational unit in LMFAO. The following layers construct an execution plan for each view group at a node. This plan needs one pass over the join of the relation at that node and the incoming views. This is yet another instance of sharing in LMFAO: The computation of different views share the scan of the relation at the node.

The Variable Order step computes a total order of the join variables in a view group. The join for a view group then sees the relation and incoming views organized logically as tries: first grouped by the first variable in the order, then by the next in the context of values for the first, and so on. This trie organization is reminiscent of factorized databases [101] and LeapFrog TrieJoin [133] and can visit up to three times less values than a standard row-based scan for our datasets.

The variable order forms the backbone of the multi-output execution plan for one view group. In the following optimization steps, we decompose the computation of views in the view group and their aggregates over this order. This results in a factorized evaluation of the queries, reminiscent of factorized aggregate evaluation algorithms [23, 101, 10].

The View Registration step identifies, for each view in a the view group, the highest variable in the order at which we can construct and output tuples to the output for this view. This optimization minimizes the number of times each view is updated, and depends

on the group-by variables of the view.

Similarly, Aggregate Registration decomposes the computation of all aggregates for the views in the group into partial aggregates, where the evaluation of each partial aggregate is registered to different variables in the order. This execution strategy is designed to maximize the computation sharing across many aggregates with different group-by and UDAFs via the introduction of local variables, and to minimize the number of accesses (initialization, update, lookup) to these local variables.

The previous two optimization steps construct a register of the computation required by the view group. This register defines the multi-output execution plan for a view group. LMFAO then generates optimized C++ code for each multi-output plan, which is specialized to the information provided by the database schema, the join tree, and the register of view and aggregate computations over the variable order. We generate separate code for each view group and also for general tasks such as data loading, which then are compiled in parallel. The code generation adopts the following low-level code optimizations.

For the Data Structure step, we select for each view a data structure that represents the result of the view. Each view is represented either by a list or a hash map, depending on which data structure allows for more efficient updates. The layer also organizes all aggregates in a contiguous fixed-size array that is ordered to allow sequential read/write.

The Loop Synthesis optimization then synthesizes loops from long sequences of lock-step aggregate computations. The organization of aggregates allows us to manage them in contiguous batches. This is reminiscent of vectorization [140], now applied to aggregates instead of data records.

During the code generation for the multi-output execution plan, LMFAO knows which joins and UDAFs are required by the workload. The Inline Function Calls optimization step thus inlines the function calls for both the join and aggregate function to reduce the interpretation overhead of the execution plan.

Some applications require the computation of UDAFs that change between iterations depending on the outcome of computation. For instance, the nodes in a decision tree are iteratively constructed in the context of conditions that are selected based on the data. These functions are tagged as *dynamic* to instruct LMFAO to avoid inlining their calls and instead generate separate code that is compiled between iterations and linked dynamically.

Finally, the Parallelization step addresses task and domain parallelism. LMFAO parallelizes the computation of multi-output plans for view groups that do not depend on each other. For this, it computes the dependency graph of the view groups. LMFAO partitions the largest input relations and allocates a thread per partition to compute the multi-output plan on that partition. This layer brought $1.4 - 3\times$ extra speedup on a machine with four vCPUs (AWS d2.xlarge).

We next discuss key design choices for LMFAO and motivate them using examples. We also present pseudocode for the main optimization steps.

Chapter 12

View Generation

We next discuss how LMFAO decomposes batches of aggregate queries into views over a join tree. In Section 12.1, we first recall the definition of join trees, and then exemplify how a single aggregate query can be evaluated over a join tree by decomposing it into views. Section 12.3 then exemplifies the importance of computing queries at different roots in the join tree, and Section 12.4 presents the pseudocode of the decomposition of arbitrary query batches into views. Finally, we exemplify in Section 12.5 how LMFAO can share and merge views across different aggregate queries in the query batch, which effectively minimizes the number of views that are computed.

12.1 Join Trees

LMFAO evaluates a batch of aggregate queries of the form (4.1) over a single join tree of the database schema, or equivalently of the natural join of the database relations. We next recall the notion of join trees and exemplify the evaluation of aggregates over joins trees by decomposing them into views.

The *join tree* of the natural join of the database relations $R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m})$ is an undirected tree T such that [3]:

- The set of nodes of T is $\{R_1, \ldots, R_m\}$.
- For every pair of nodes R_i and R_j, their common variables are in the schema of every node R_k along the distinct path from R_i to R_j, i.e., $\omega_{R_i} \cap \omega_{R_j} \subseteq \omega_{R_k}$.

Acyclic joins always admit join trees. Arbitrary joins are transformed into acyclic ones by means of hypertree decompositions and materialization of their nodes (called bags) using worst-case optimal join algorithms [84, 133].

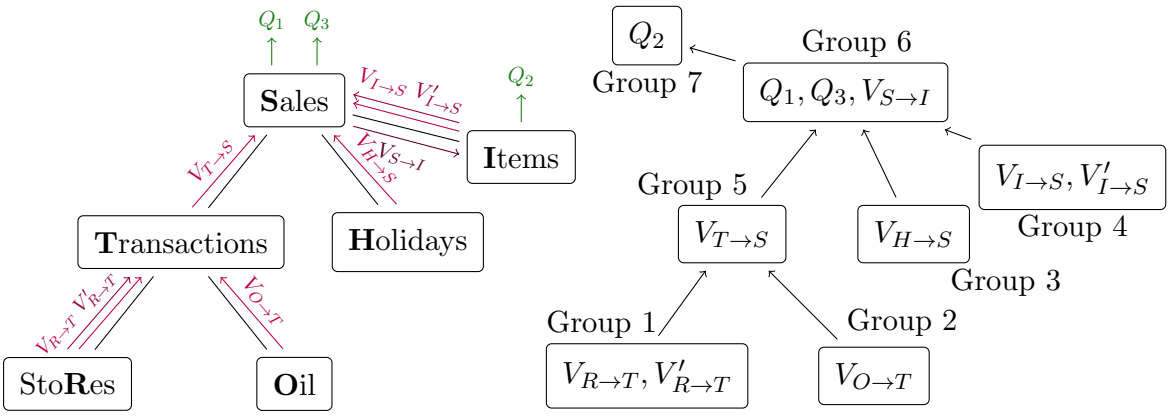


Figure 12.1: (left) A join tree for the schema of the Favorita dataset from Example 12.1 with directional views and four queries, partitioned in 7 groups. (right) The dependency graph of the view groups.

Example 12.1. The Favorita dataset [49] has the following six relations:

Sales (<u>date</u>, <u>store</u>, <u>item</u>, units, promo)

Stores (<u>store</u>, city, state, stype, cluster)

Items (<u>item</u>, family, class, perishable)

Transactions (<u>date</u>, <u>store</u>, transactions)

Holidays (<u>date</u>, htype, locale, transferred)

Oil (<u>date</u>, price)

Sales relation keeps, for each store, item, and date, the number of units sold and whether or not this item was on promotion. **Stores** provides for each store the city and state where it is located in, the type of the stores, and the cluster it belongs to. **Items** keeps, for each item, the item family and item class it belongs to, and whether or not it is perishable. **Transactions** keeps the number of transactions of each store at a given date. **Holidays** indicates whether a date is a holiday, and if so the location where it is observed and whether or not it has been transferred. Finally, **Oil** provides the oil price for each date.

Figure 12.1(left) shows a possible join tree for the natural join of the six relations. Instead of showing join variables on the edges, we underline them in the schema to avoid clutter. We use this join tree to exemplify the optimization steps performed by LMFAO. □

12.2 Computing Aggregates over Join Trees

LMFAO computes aggregates over join trees by decomposing the queries into views [23, 10]. Consider a join tree Δ with root R and children $C_1, \ldots, C_k$ of R, where child C_i is the root of a subtree Δ_i in Δ. Let ω_{Δ_i} denote the union of the schemas of all relations in Δ_i. A query

$Q(F, \texttt{SUM}(\alpha))$ with group-by variables F and aggregate expression α is decomposed over Δ into a view that is computed over the join of R and views computed over each subtree Δ_i:

$$Q(F, \texttt{SUM}(\alpha)) \leftarrow R(\omega_R), V_{C_1}(F_1, \alpha_1), \ldots, V_{C_k}(F_k, \alpha_k) \tag{12.1}$$

The view $V_{C_i}(F_i, \alpha_i)$ for a child C_i of R is defined as the "projection" of Q onto Δ_i. Its body is the natural join of the relations in Δ_i. The group-by variables F_i of V_{C_i} is the subset of ω_{Δ_i} that is required to support the aggregate α, the set F, and the join between C_i and R. The aggregate α_i is the partition of α that can be evaluated over Δ_i. We can decompose the views V_{C_i} recursively as explained for Q.

Example 12.2. Let us compute the sum of the product of two aggregate functions $f(\text{units})$ and $p(\text{price})$ over the natural join of the Favorita relations:

$$Q_1(\texttt{SUM}(f(\text{units}) \cdot p(\text{price}))) \leftarrow S(\omega_S), T(\omega_T), R(\omega_R), O(\omega_O), H(\omega_H), I(\omega_I)$$

We abbreviated the names of the Favorita relations as highlighted in Figure 12.1. The aggregate functions f and p are over the variables units in Sales and price in Oil. We can rewrite Q_1 to push these functions down to the relations and also follow the structure of the join tree in Figure 12.1(left):

$$\begin{aligned}
V_O(\text{date}, \texttt{SUM}(p(\text{price}))) &\leftarrow O(\text{date}, \text{price}) \\
V_R(\text{store}, \texttt{SUM(1)}) &\leftarrow R(\text{store}, \text{city}, \text{state}, \text{stype}, \text{cluster}) \\
V_T(\text{date}, \text{store}, \texttt{SUM}(c \cdot p)) &\leftarrow T(\text{date}, \text{store}, \text{transactions}), V_R(\text{store}, c), V_O(\text{date}, p) \\
V_H(\text{date}, \texttt{SUM(1)}) &\leftarrow H(\text{date}, \text{htype}, \text{locale}, \text{transferred}) \\
V_I(\text{item}, \texttt{SUM(1)}) &\leftarrow I(\text{item}, \text{family}, \text{class}, \text{perishable}) \\
Q_1(\texttt{SUM}(f(\text{units}) \cdot p \cdot c_1 \cdot c_2)) &\leftarrow V_T(\text{date}, \text{store}, p), V_I(\text{item}, c_2), V_H(\text{date}, c_1), \\
&\quad\; S(\text{date}, \text{store}, \text{item}, \text{units})
\end{aligned}$$

Only V_S and V_O have variables in the aggregate functions. For V_R, V_H, and V_I, we only need to count the number of tuples in the underlying relation. For V_T, we aggregate over the product of the aggregates from the incoming views V_R and V_O. □

The computation of several aggregate queries over the same join tree may share views between them.

Example 12.3. Consider now the query over the same join:

$$Q_2(\text{family}, \texttt{SUM}(f(\text{units}) \cdot p(\text{price}) \cdot g(\text{item}))) \leftarrow S(\omega_S), T(\omega_T), R(\omega_R), O(\omega_O), H(\omega_H), I(\omega_I)$$

This query reports the sum over the product of functions $f(\text{units})$, $p(\text{price})$, and $g(\text{item})$

for each item family. We can rewrite it similarly to Q_1 in Example 12.2:

$$V'_I(\text{item}, \text{family}, \texttt{SUM(1)}) \leftarrow I(\text{item}, \text{family}, \text{class}, \text{perishable})$$
$$Q_2(\text{family}, \texttt{SUM(}f(\text{units}) \cdot g(\text{item}) \cdot p \cdot c_1 \cdot c_2\texttt{)}) \leftarrow V_T(\text{date}, \text{store}, p), V'_I(\text{item}, \text{family}, c_2), V_H(\text{date}, c_1), S(\text{date}, \text{store}, \text{item}, \text{units})$$

where the views V_T, V_O, V_R, and V_H are shared between Q_1 and Q_2. □

The complexity of computing multiple aggregate queries with different group-by variables is the lowest if we choose for each query an optimal join tree, where the group-by variables form a connected subtree. At the same time, we would like to exploit the fact that all queries are computed over the same join, and share as much computation between aggregates as possible. This is intuitively maximized if all queries are computed over the same join tree. We next discuss our solution to this tension between complexity and sharing.

12.3 Find Roots: Each Aggregate to Its Own Root

When multiple aggregates are evaluated over the same join, then the overall complexity of the evaluation of the aggregate batch may be lowered by using different roots for different queries. To accommodate for this evaluation approach, we introduce *directional views*, which are queries of the form (4.1) where we also specify their direction. They flow along an edge in the join tree from a source node to a neighboring target node, and are computed over the join of the relation at the source node and some of its incoming views. In Equation (12.1), the incoming view V_{C_i} for each child C_i of R has the direction $C_i \rightarrow R$. In the following, we denote the view V_{C_i} by $V_{C_i \rightarrow R}$ to specify its direction.

Queries Q_1 and Q_2 from Examples 12.2 and 12.3 are decomposed into views whose directions are always towards the root Sales. In the following example, we provide an alternative of evaluation strategy for Q_2.

Example 12.4. Consider query Q_2 in Example 12.3. An alternative evaluation for Q_2 would not create the view $V'_{I \rightarrow S}(\text{item}, \text{family}, \texttt{SUM(1)})$ and instead create a view

$$V_{S \rightarrow I}(\text{item}, \texttt{SUM(}f(\text{units}) \cdot g(\text{item}) \cdot p \cdot c\texttt{)}) \leftarrow S(\omega_S), V_{T \rightarrow S}(\text{date}, \text{store}, p), V_{H \rightarrow S}(\text{date}, c) \tag{12.2}$$

Then, we can compute Q_2 over the join of Items and $V_{S \rightarrow I}$:

$$Q_2(\text{family}, \texttt{SUM(}p\texttt{)}) \leftarrow I(\text{item}, \text{family}, \text{class}, \text{perishable}), V_{S \rightarrow I}(\text{item}, p) \tag{12.3}$$

The advantage of this rewriting is that we do not need to construct a view with group-by variables that are not join variables (e.g., $V'_{I \rightarrow S}$ has the additional group-by variable family), because additional group-by variables can significantly increase the size of the view.

The queries Q_1 and Q_2 are thus computed over the same join tree but rooted at different nodes: Sales for Q_1 and Items for Q_2. This also means that the edge between Sales and Item has two views, yet with different direction. □

Using different roots for different queries may lower the overall complexity of evaluating a batch of aggregates. We next exemplify the advantage of evaluating aggregates at different roots in the join tree for a batch of queries that is common in linear regression and mutual information settings where all variables are categorical. We then explain how to find a root for a given aggregate in a batch of aggregates.

Example 12.5. Consider the following batch of count queries over the join of relations $S_k(X_k, X_{k+1})$ of size N, $\forall k \in [n-1]$:

$$\forall i \in [n] : Q_i(X_i, \texttt{SUM(1)}) \leftarrow S_1(X_1, X_2), \ldots, S_{n-1}(X_{n-1}, X_n)$$

We first explain how to compute these n queries by decomposing them into directional views that are over the join tree $S_1 - S_2 - \cdots - S_{n-1}$ with root S_1 and have the same direction along this path towards the root.

For simplicity, we denote by L_k^i the view constructed for Q_i with direction from S_k to S_{k-1}. The views are defined as follows, with the addition of $L_n^n(X_n, X_n, \texttt{SUM(1)}) \leftarrow \mathrm{Dom}(X_n)$ that associates each value in the domain of X_n with 1.

$$\begin{aligned}
&\forall k \in [i-1] : \; L_k^i(X_k, X_i, \texttt{SUM}(c)) \leftarrow S_k(X_k, X_{k+1}), L_{k+1}^i(X_{k+1}, X_i, c) \\
&\forall i \in [n-1] : \; L_i^i(X_i, X_i, \texttt{SUM}(c)) \leftarrow L_{i+1}^{i+1}(X_i, X_{i+1}, c) \\
&\forall i \in [n-1] : \qquad Q_i(X_i, \texttt{SUM}(c)) \leftarrow L_1^i(X_1, X_i, c)
\end{aligned}$$

The above decomposition proceeds as follows. Q_n counts the number of occurrences of each value for X_n in the join. We start with 1, as provided by L_n^n, and progress to the left towards the root S_1. The view L_{n-1}^n computes the counts for X_n in the context of each value for X_{n-1} as obtained from S_{n-1}. We need to keep the values for X_{n-1} to connect with S_{n-2}. Eventually, we reach L_1^n that gives the counts for X_n in the context of X_1, and we sum them over the values of X_1. The same idea applies to any Q_i with one additional optimization: Instead of starting from the leaf S_{n-1}, we can jump-start at S_{i-1} and reuse the computation of the counts for X_{i+1} in the context of X_i as provided by L_{i+1}^{i+1}. We need $O(n^2)$ many views and those of them that have group-by variables from two different relations take $O(N^2)$ time.

We can lower this complexity to $O(N)$ by using different roots for different queries. We show the effect of using the root S_i for query Q_i. For each query Q_i, we construct two directional views: view R_i from S_{i-1} to S_i (i.e., from left to right) and view L_i from S_{i+1} to S_i (i.e., from right to left). The counts for X_i values are the products of the counts in

the left view L_i and the right view R_i:

$$\begin{aligned}
\forall i \in [n] : L_i(X_i, \texttt{SUM(}c\texttt{)}) &\leftarrow S_i(X_i, X_{i+1}), L_{i+1}(X_{i+1}, c) \\
\forall i \in [n] : R_i(X_i, \texttt{SUM(}c\texttt{)}) &\leftarrow S_{i-1}(X_{i-1}, X_i), R_{i-1}(X_{i-1}, c) \\
\forall i \in [n] : Q_i(X_i, \texttt{SUM(}c_1 \cdot c_2\texttt{)}) &\leftarrow L_i(X_i, c_1), R_i(X_i, c_2)
\end{aligned}$$

We also use two trivial views $R_1(X_1, 1) \leftarrow \mathrm{Dom}(X_1)$ and $L_n(X_n, 1) \leftarrow \mathrm{Dom}(X_n)$. Note how the left view L_i is expressed using the left view L_{i+1} coming from the node S_{i+1} below S_i. Similarly for the right views. Each of the $2n$ views takes linear time. Moreover, they share much more computation among them than the views L_k^i used in the first scenario.

The second approach that chooses the root S_i for query Q_i can also be used for queries over all pairs of variables:

$$\forall i, j \in [n] : Q_{i,j}(X_i, X_j, \texttt{SUM(1)}) \leftarrow S_1(X_1, X_2), \ldots, S_{n-1}(X_{n-1}, X_n)$$

Each of these n^2 queries takes time $O(N)$ for $|i - j| \leq 1$ and $O(N^2)$ otherwise. At each node S_i, we compute a left view $L_{i,j}$, for any $i < j \leq n$, that counts the number of tuples for (X_i, X_j) over the path $S_i - \cdots - S_{n-1}$. Then, the overall count c in $Q_{i,j}(X_i, X_j, \texttt{SUM(}c\texttt{)})$ is computed as the product of the count for X_i given by the right view R_i and the count for (X_i, X_j) given by the left view $L_{i,j}$ ($\forall 1 \leq i < j < n$):

$$\begin{aligned}
L_{i,j}(X_i, X_j, \texttt{SUM(}c\texttt{)}) &\leftarrow S_i(X_i, X_{i+1}), L_{i+1,j}(X_{i+1}, X_j, c) \\
Q_{i,j}(X_i, X_j, \texttt{SUM(}c_1 \cdot c_2\texttt{)}) &\leftarrow L_{i,j}(X_i, X_j, c_1), R_i(X_i, c_2)
\end{aligned}$$

The views $\forall i \in [n] : L_{i,i}(X_i, X_i, 1) \leftarrow \mathrm{Dom}(X_i)$ assign a count of 1 to each value of X_i. □

Using different roots for different queries may lower the overall complexity of evaluating a batch of aggregates. At the same time, we would like to share computation as much as possible, which is intuitively maximized if all queries are computed at the same root. We next discuss our solution to this tension between complexity and sharing.

The goal is to choose a root in the join tree for each query so that the size of all views computed for the query batch is minimized, where the size depends on both the number of tuples as well as the arity of the view. LMFAO uses a simple but effective approximation of this problem, which depends on whether the query has group-by variables or not.

We first consider all queries in the batch that have group-by variables. For each such query Q, we assign a weight to each relation R in the join tree that is the fraction of the number of group-by variables of Q in R. At the end of this weight assignment phase, each relation will be assigned some weight. We assign roots in the decreasing order of their weights. A relation with the largest weight is then assigned as root to all queries that considered it as possible root. We break ties by choosing a relation with the largest size. The rationale for this choice is threefold. The choice for the largest relation avoids the

creation of possibly large views over it. If the root for Q has no group-by variable of Q, then we will create views carrying around values for these variables, which results in larger views. A root with a large weight ensures that many views share the same direction towards it, so their computation may be shared and they may be merged or grouped (as explained in the next sections).

For each query Q in the batch that has no group-by variables, the root assignment depends in the variables in the aggregate expression α. Recall that α defines a sum of products of functions. For each product $P \in \alpha$, we collect the set ω_P of variables that is the union over the variable sets for all functions in P. We then assign a weight to each relation R in the join tree that is the fraction of the number of variables ω_P in R. The root for Q is the relation that has the largest weight over all products in α, where ties are again broken by choosing the largest relation. The rationale for this heuristic is the following: any function in α that is evaluated over a non-root relation in the join tree is propagated to the root in form of an aggregate in the views. By choosing as root the relation that supports most functions in α, the incoming views need to compute less aggregates, which in turn decreases the arity of the view.

12.4 Aggregate Pushdown over Join Trees

We next present the pseudocode for the decomposition of arbitrary queries of the form (4.1) into views over a given join tree Δ. The procedure is presented in two parts. We first present and exemplify Algorithm 12.1, which decomposes an aggregate expression `SUM(`α`)` into partial aggregates, such that each partial aggregate can be computed over different subtrees of Δ. We then present Algorithm 12.2, which decomposes the queries into views over each subtree of Δ, where each view computes one partial aggregate given by Algorithm 12.1 over the corresponding subtree of Δ.

12.4.1 Decomposing aggregates into lists of partial aggregates

We consider an aggregate expression `SUM(`α`)`, where α denotes a sum of products of functions as defined by (4.3). LMFAO represents this expression as a list P_{list} of products, where each product $P \in P_{\text{list}}$ defines a list of functions $[f : \mathcal{V} \to \mathbb{R}]_{f \in P}$. We use $+\!\!+$ to denote the list concatenation operator.

Algorithm 12.1 takes as input a list P_{list} of products and a rooted join tree Δ. We assume, without loss of generality, that the root of Δ is the relation R with schema ω_R, and it has k children that are the root for subtrees $(\Delta_c)_{c \in [k]}$ in Δ. The algorithm has two steps: (1) we first decompose each product $P \in P_{\text{list}}$ into partial products that can be computed over different subtrees in Δ, and then (2) we seek partial products that can be merged into partial aggregates that are summations over partial products. The algorithm returns a list D_{list}, where each $D \in D_{\text{list}}$ is a list of partial aggregates.

decomposeAggregate (Product List P_{list}, Rooted Join Tree Δ)

let : $P_{\text{list}} = [P_1, \ldots, P_s]$ where $\forall j \in [s] : P_j = [\ f_{j1}(\omega_{f_{j1}}), \ldots, f_{jp}(\omega_{f_{jp}})\ ]$
let : $\Delta = R(\Delta_1, \ldots, \Delta_k)$;

$D_{\text{list}} = [\ ]$ $\quad \forall c \in [k] : \ \omega_{\Delta_c} = \bigcup_{S \in \mathsf{nodes}(\Delta_c)} \omega_S$;
foreach $P \in P_{\text{list}}$ **do** {
  /* Step 1: Decompose Product P into Partial Products */
  $\omega = \emptyset$; $\quad \alpha_\Delta = [\ ]$; $\quad \forall c \in [k] : \ \alpha_{\Delta_c} = [\ ]$;
  foreach $f_j(\omega_{f_j}) \in P$ {
    switch(ω_{f_j}) {
      case $(\omega_{f_j} \subseteq \omega_R)$: $\alpha_\Delta = \alpha_\Delta +\!\!+ f_j(\omega_{f_j})$;
      case $(\exists c \in [k] : \omega_{f_j} \subseteq \omega_{\Delta_c})$: $\alpha_{\Delta_c} = \alpha_{\Delta_c} +\!\!+ f_j(\omega_{f_j})$;
      default : { $\alpha_\Delta = \alpha_\Delta +\!\!+ f_j(\omega_{f_j})$; $\quad \omega \ \cup\!= \omega_{f_j} \setminus \omega_R$; };
    }
  }
  /* Step 2: Merge Decomposed Products into Sums of Partial Products */
  if $(\exists\, [\omega', \alpha'_\Delta, \alpha'_{\Delta_1}, \ldots, \alpha'_{\Delta_k}] \in D_{\text{list}})$ s.t. $(\alpha_\Delta = \alpha'_\Delta \wedge \exists j \in [k] : \forall_{i \neq j \in [k]} \alpha_{\Delta_i} = \alpha'_{\Delta_i})$
    $\alpha'_{\Delta_j} = \alpha'_{\Delta_j} +\!\!+ [\alpha_{\Delta_j}]$;
  else $D_{\text{list}} = D_{\text{list}} +\!\!+ [\ \omega, \alpha_\Delta, [\alpha_{\Delta_1}], \ldots, [\alpha_{\Delta_k}]\ ]$;
}
return D_{list};

Algorithm 12.1: Decomposition of aggregates that are sums of products of functions into partial aggregates that can be computed over different subtrees in the join tree Δ. The aggregate is represented as a list P_{list} of products, where each product $P \in P_{\text{list}}$ is a list of functions. We use R to denote the root of Δ, which has k children that are roots of subtrees $(\Delta_c)_{c \in [k]}$. We use $+\!\!+$ to denote list concatenation.

We next explain and exemplify the first step of the algorithm. The algorithm considers each product $P \in P_{\text{list}}$, and decomposes it into $k+1$ disjoint lists $[\alpha_\Delta, \alpha_{\Delta_1}, \ldots, \alpha_{\Delta_k}]$. Each list $(\alpha_{\Delta_j})_{j \in [k]}$ captures the functions in P that can be computed over subtree Δ_j. The list α_Δ keeps the functions in P that need to be computed over the join of relation R and incoming views from each subtree Δ_j. In addition, the algorithm computes a set ω of variables, which contains all variables that are needed to evaluate the functions in α_Δ.

The partial product lists and the variable set ω are initially empty. Each function $[f(\omega_f)]_{f \in P}$ is added to one partial product list depending on the following three cases:

(1) If ω_f is a subset of ω_R, this function can be evaluated directly over R and we add it to α_Δ. If ω_f is covered by several relations in the join tree, this case ensures that a function is evaluated only once.

(2) If ω_f is not covered by ω_R, then the algorithm checks if there exists a subtree Δ_c such that the union of the schemas of the relations in the subtree (denoted by ω_{Δ_c}) contains all variables in ω_f. If so, the function f is added to α_{Δ_c}.

(3) If there is no subtree that covers ω_f, the function $f(\omega_f)$ must be computed over the

join of relation R and incoming views. Thus, we add f to α_Δ. In this case, we also add to ω all variables in $\omega_f \setminus \omega_R$, which are the variables that are required to evaluate f but not supported by relation R.

Example 12.6. Recall the join tree Δ for the Favorita dataset from Example 12.1. The root of Δ is Sales, which has three subtrees:

$$\Delta_T = \text{(Transactions (Stores, Oil))} \qquad \Delta_H = \text{(Holiday)} \qquad \Delta_I = \text{(Items)}$$

The following query computes an aggregate that is defined by a product of three functions:

$$Q(\texttt{SUM}(f(\text{units}) \cdot p(\text{price}) \cdot h(\text{class, locale}))) \leftarrow S(\omega_S), T(\omega_T), R(\omega_R), O(\omega_O), H(\omega_H), I(\omega_I)$$

Each function f, p, and h is registered to one of the partial products α_Δ, α_{Δ_T}, α_{Δ_H}, and α_{Δ_I}, depending on the three cases outlined above.

For f(units), the variable units is in ω_S, and thus f(units) is assigned to α_Δ. Note that f could also be evaluated over Δ_T or Δ_H, but we register it to α_Δ to ensure that it is only evaluated once.

For function p(price), the variable price is not in the schema of Sales, but it occurs in the subtree Δ_T. Therefore, p(price) is assigned to α_{Δ_T}.

The variables of h(class, locale) are covered neither by Sales nor any of the subtrees. Therefore, h must be evaluated over the join of Sales and incoming views from each subtree, and we add it to α_Δ. In addition, the algorithm adds variables class and locale to ω_P, indicating that the views from the subtrees need to propagate these two variables so that the join of Sales and the incoming views can support the evaluation of h.

The product of f, p, and h is thus decomposed over Δ into the partial products:

$$\alpha_\Delta = [f(\text{date}), h(\text{class, locale})] \qquad \alpha_{\Delta_T} = [p(\text{price})] \qquad \alpha_{\Delta_H} = [\,] \qquad \alpha_{\Delta_I} = [\,]$$

and $\omega = \{\text{class, locale}\}$. □

Following the decomposition of the product into partial products, the algorithm checks if the partial products can be merged with any other previously decomposed aggregate in D_{list}. Let $D = [\omega', \alpha'_\Delta, \alpha'_{\Delta_1}, \ldots, \alpha'_{\Delta_k}]$ denote a list of partial aggregates in D_{list}. The decomposed product for P can be merged with D, if and only if $\alpha_\Delta = \alpha'_\Delta$, and there exists one subtree Δ_i, such that for all other subtrees $(\Delta_j)_{j \in [k], j \neq i}$, $\alpha_{\Delta_j} = \alpha'_{\Delta_j}$. If this is the case, we can concatenate α'_{Δ_i} and $[\alpha_{\Delta_i}]$, which indicates that we compute a sum of partial products over the subtree Δ_i. If the partial products for P cannot be merged with any $D \in D_{\text{list}}$, their list is added to D_{list}. Note that for each $j \in [k]$, the partial product α_{Δ_j} is turned into a list, and thus D_{list} is a list of partial aggregates. The merging of partial products results in factorized evaluation of the merged products in P_{list} over the join tree Δ as exemplified in the following example.

Example 12.7. We modify the Example 12.6 and consider the following query:

$$Q(\texttt{SUM}(\alpha)) \leftarrow S(\omega_S), T(\omega_T), R(\omega_R), O(\omega_O), H(\omega_H), I(\omega_I)$$

where $\alpha = f(\text{units}) \cdot g(\text{price}) \cdot h(\text{class}) + f(\text{units}) \cdot g(\text{price}) \cdot h'(\text{perishable})$.

LMFAO encodes the aggregate α as a list of two products:

$$P = [f(\text{units}), g(\text{price}), h(\text{class})] \qquad P' = [f(\text{units}), g(\text{price}), h'(\text{perishable})]$$

The two products can be decomposed into the following two lists of partial products:

$$\begin{array}{lllll} \omega = \emptyset & \alpha_\Delta = [f(\text{units})] & \alpha_{\Delta_T} = [g(\text{price})] & \alpha_{\Delta_H} = [h(\text{class})] & \alpha_{\Delta_I} = [\,] \\ \omega' = \emptyset & \alpha'_\Delta = [f(\text{units})] & \alpha'_{\Delta_T} = [g(\text{price})] & \alpha'_{\Delta_H} = [h'(\text{perishable})] & \alpha'_{\Delta_I} = [\,] \end{array}$$

Note that all partial products except the ones for Δ_H are identical. Thus, we can merge the two lists into a single list, where α_{Δ_H} now encodes a list of partial products, or equivalently a sum over the two partial products:

$$\omega = \emptyset \quad \alpha_\Delta = [f(\text{units})] \quad \alpha_{\Delta_T} = [g(\text{price})] \quad \alpha_{\Delta_H} = [[h(\text{class})], [h'(\text{perishable})]] \quad \alpha_{\Delta_I} = [\,]$$

The merging of the two list is equivalent to a factorization of α:

$$\alpha = f(\text{units}) \cdot g(\text{price}) \cdot (h(\text{class}) + h'(\text{perishable})). \tag{12.4}$$

The result of this merge is that we can now push to Δ_H an aggregate that is the sum of the functions h and h', instead of pushing two individual aggregates for each function. This means that we compute less views over Δ_H. □

The merging of decomposed products is particularly useful for the computation of the gradients for regression problems, which requires queries where the aggregates are sums over all features (cf. Section 6.3).

12.4.2 Pushdown of Decomposed Aggregates

We next present how LMFAO decomposes aggregate queries of the form (4.1) into views over the join tree Δ. The procedure is presented in Algorithm 12.2.

The algorithm takes as input a query Q and a join tree Δ, which is rooted at R and has k children $(C_j)_{j\in[k]}$ that are the roots for subtrees $(\Delta_j)_{j\in[k]}$ in Δ. The algorithm returns a view that represents Q and is computed over the join of R and incoming views from each subtree Δ_j. Without loss of generality we assume that Q computes only a single aggregate, which is defined by a list P_{list} of products. If a query has multiple aggregates, we can call the **aggregatePushdown** method for each individual aggregate, and then merge the views

aggregatePushdown (Query $Q(\omega_Q, \texttt{SUM}(P_{list}))$, Rooted Join Tree Δ)

let : $\Delta = R(\Delta_1, \ldots, \Delta_k)$;
let : $P_{\text{list}} = [P_1, \ldots, P_s]$; /* P_{list} is a list of products; */

if $(Q \in \mathsf{cache}_\Delta)$ **return** $\mathsf{cache}_\Delta(Q)$;

$T = \mathsf{parent}(\Delta)$; $\quad \forall j \in [k] : \omega_{\Delta_j} = \bigcup_{S \in \mathsf{relations}(\Delta_j)} \omega_S$; $\quad C_j = \mathsf{root}(\Delta_j)$;

$D_{\text{list}} = \textbf{decomposeAggregate}(P_{\text{list}}, \Delta)$;

foreach $i \in [\,|D_{\text{list}}|\,]$ where $D_{\text{list}}[i] = [\,\omega, \alpha_\Delta, \alpha_{\Delta_1}, \ldots, \alpha_{\Delta_k}\,]$ **do** {
  foreach $j \in [k]$ **do**
    $V_{C_j \to R}(\omega_j, \alpha_j) = \textbf{aggregatePushdown}(Q_j((\omega \cup \omega_R \cup \omega_Q) \cap \omega_{\Delta_j}, \texttt{SUM}(\alpha_{\Delta_j})),\ \Delta_j)$;
  $V^{(i)}_{R \to T}(\omega_Q, \texttt{SUM}(\alpha_\Delta \cdot \alpha_1 \cdot \ldots \cdot \alpha_k)) = R(\omega_R), V_{C_1 \to R}(\omega_1, \alpha_1), \ldots, V_{C_k \to R}(\omega_k, \alpha_k)$;
}
$V_{R \to T}(\omega_Q, \alpha_1 + \ldots + \alpha_{|D_{\text{list}}|}) = V^{(1)}_{R \to T}(\omega_Q, \alpha_1), \ldots, V^{(|D_{\text{list}}|)}_{R \to T}(\omega_Q, \alpha_{|D_{\text{list}}|})$;
$\mathsf{cache}_\Delta(Q) = V_{R \to T}$;
return $V_{R \to T}$;

Algorithm 12.2: Pseudocode of the aggregate pushdown optimization step in LMFAO. Each input query Q is decomposed into views over the join tree Δ, which is rooted at R and has k children $(C_i)_{i \in [k]}$. The aggregate in Q is defined by a list P_{list} of products of user defined aggregate functions.

for each aggregate into a single view as described in Section 12.5. Let relation T be the parent of R, or $\emptyset$ if R is the root for query Q and does not have a parent.

The algorithm first probes cache_Δ to check if this particular aggregate query has been decomposed over the join tree Δ before. If so, it returns the previously defined view. Otherwise, it decomposes the aggregate by passing P_{list} and Δ to the **decomposeAggregate** function from Algorithm 12.1, which returns a list D_{list} of partial aggregates.

For each $i \in [|D_{\text{list}}|]$, the algorithm recurses to each subtree $(\Delta_j)_{j \in [k]}$ and gets a view $V_{C_j \to R}$. The input to the recursive call is a query Q_j and subtree Δ_j. Let ω_{Δ_j} denote the schema of Δ_j. The group-by variables of Q_j are given by the union of the following sets:

(1) The intersection between the schema ω_R of relation R and ω_{Δ_j}, which by definition is the set of join variables between R and C_j.

(2) The intersection of the group-by variables ω_Q of Q and ω_{Δ_j}.

(3) The intersection of the variable set ω in $D_{\text{list}}[i]$ and the schema of Δ_j, which defines the subset of ω_{Δ_j} that is required to compute the functions in α_Δ over the join of relation R and the incoming views from each subtree Δ_j.

The aggregate of the query Q_j is defined by the partial aggregate α_{Δ_j} in $D_{\text{list}}[i]$.

The algorithm creates two types of views. First, we construct an "intermediate" view $V^{(i)}_{R\to T}$ for each element in D_{list}. This view is computed over the join of R and the incoming views $V_{C_j\to R}$ from each child $(C_j)_{j\in[k]}$ of R:

$$V^{(i)}_{R\to T}(\omega_Q, \texttt{SUM}(\alpha_\Delta \cdot \alpha_1 \cdot \ldots \cdot \alpha_k)) = R(\omega_R), V_{C_1\to R}(\omega_1, \alpha_1), \ldots, V_{C_k\to R}(\omega_k, \alpha_k) \tag{12.5}$$

The group-by variables of $V_{R\to T}$ are given by group-by variables ω_Q of query Q. The aggregate is defined by the product of α_Δ in $D_{\text{list}}[i]$ and aggregates from incoming views.

For the second type of views, we consolidate the intermediate views $(V^{(i)}_{R\to T})_{i\in[|D_{\text{list}}|]}$ into a single "outgoing" view $V_{R\to T}$:

$$V_{R\to T}(\omega_Q, \alpha_1 + \ldots + \alpha_{|D_{\text{list}}|}) = V^{(1)}_{R\to T}(\omega_Q, \alpha_1), \ldots, V^{(|D_{\text{list}}|)}_{R\to T}(\omega_Q, \alpha_{|D_{\text{list}}|}) \tag{12.6}$$

The aggregate of $V_{R\to T}$ is defined by the summation over the aggregates for each intermediate view $(V^{(j)}_{R\to T})_{j\in[k]}$. The view $V_{R\to T}$ precisely represents the aggregate query Q. The algorithm adds it to the cache for Δ and then returns it.

Intermediate views of the form (12.5) are only represented at the syntactic level. During the multi-output optimization layer, the computation of intermediate views is inlined with the computation of the outgoing view $V_{R\to T}$, so that only one view is computed per query and edge in the join tree.

12.5 Merging Views

The views generated by Algorithm 12.2 can be consolidated or *merged*, which is beneficial for two reasons: (1) We can share computation across different views, and (2) we can lower the number of views that are computed. We differentiate between three cases of merging:

- Case (1) merges views that share the same direction, group-by variables, aggregate, and join body. That is, the same view is created for several queries. In this case we only need to compute the view once, and then reuse it for each query that requires this view. This case is already captured by the cache in Algorithm 12.2, and is exemplified in Example 12.4.

- Case (2) concerns views with the same direction, group-by variables, and join, but different aggregates. Such views are merged into a single view that keeps the same group-by variables and join but merges the lists of aggregates.

- Case (3) merges views that have the same direction and group-by variables, but not the same join body and aggregate. This is the most general form of merging supported by LMFAO. The consolidated view is a new view that is a join of these views on their (same) group-by variables (as shown for outgoing views of the form (12.6)), which then have multiple aggregates in the head.

We next exemplify the merging of views for Cases (2) and (3).

Example 12.8. We continue Examples 12.2 and 12.3 and add a third query over the same join body as Q_1 and Q_2:

$$Q_3(\text{store}, \texttt{SUM(}k(\text{txns,city}) \cdot g(\text{item}) \cdot h(\text{date,family})\texttt{)}) \leftarrow S(\omega_S), T(\omega_T), R(\omega_R), O(\omega_O), H(\omega_H), I(\omega_I)$$

This is decomposed into the following views over the same join tree rooted at Sales:

$$\begin{aligned}
V'_O(\text{date}, \texttt{SUM(1)}) &\leftarrow O(\text{date}, \text{price}) \\
V'_R(\text{store}, \text{city}, \texttt{SUM(1)}) &\leftarrow R(\text{store}, \text{city}, \text{state}, \text{stype}, \text{cluster}) \\
V'_T(\text{date}, \text{store}, \texttt{SUM(}k(\text{txns}, \text{city}) \cdot p \cdot q\texttt{)}) &\leftarrow T(\text{date}, \text{store}, \text{txns}), V'_R(\text{store}, \text{city}, p), V'_O(\text{date}, q) \\
Q_3(\text{store}, \texttt{SUM(}g(\text{item}) \cdot h(\text{date,family}) \cdot c_1 \cdot c_2 \cdot c_3\texttt{)}) &\leftarrow S(\text{date}, \text{store}, \text{item}, \text{units}), V_H(\text{date}, c_1), \\
&\qquad V'_T(\text{date}, \text{store}, c_2), V'_I(\text{family}, \text{item}, c_3)
\end{aligned}$$

The missing view V_H is shared with Q_1 and Q_2, while V'_I was defined for Q_2 in Example 12.3.

The view $V_O(\text{date}, p(\text{price}))$ for Q_1 can be merged with $V'_O(\text{date}, 1)$ for Q_3 into a new view W_O since they have the same group-by variables and body following Case (2):

$$W_O(\text{date}, \texttt{SUM(}p(\text{price})\texttt{)}, \texttt{SUM(1)}) \leftarrow O(\text{date}, \text{price})$$

Both views V_T for Q_1 and V'_T for Q_3 are now defined over W_O instead of the views V_O and V'_O. Views V_T and V'_T have the same group-by variables and direction, but different bodies (one joins over V_R and the other over V'_R). We can merge them in W_T following Case (3):

$$W_T(\text{date}, \text{store}, \texttt{SUM(}a_1\texttt{)}, \texttt{SUM(}a_2\texttt{)}) \leftarrow V_T(\text{date}, \text{store}, a_1), V'_T(\text{date}, \text{store}, a_2)$$

Figure 12.1(left) shows queries Q_1, Q_2, and Q_3 and their directional views along the edges of our Favorita join tree. □

Chapter 13

Multi-Output Optimization

We next detail the key design choices for the multi-output optimization layer. Algorithm 13.1 summarizes the optimization steps of this layer.

The input to the layer is a list of views that were generated by the view generation layer. We then cluster the views into groups of views that go out of the same node, regardless of their direction, group-by variables, and bodies.

A view group is a computational unit in LMFAO. The multi-output optimization layer constructs a novel execution plan that computes all views in a group in one scan over their common input relation. Since this plan outputs the results for several views, we call it the *multi-output execution plan*.

One source of complexity in the multi-output execution plan is that the views in the group are defined over different incoming views. While scanning the common relation, the multi-output plan looks-up into the incoming views to fetch aggregates needed for the computation of the views in the group. A second challenge is to update the aggregates of each view in the group as soon as possible and with minimal number of computation steps.

To address these challenges, LMFAO constructs an order of the join variables, which defines a trie interface for the underlying join and supports the join over different incoming views. Then, we decompose the aggregate computation over the variable order, by registering different components of the outgoing views to different levels in the order. As a result, we factorize the computation of all views in the group, share computation across views, and minimize operations that need to be performed for each view.

We next present each optimization step of this layer.

13.1 Grouping Views

We first consider the procedure that groups views. A view group is a list of (merged) views that are computed over the same node in the join tree. The views in a group may have different group-by variables, directions, and join bodies, but they must be evaluated at the

```
multioutputOptimization (View List V_list)
G_list = groupViews(V_list)
foreach G ∈ G_list do {
  Λ = variableOrder(G);
  foreach V ∈ G do { registerView(V, Λ); }        /* V is an outgoing view */
}
```

Algorithm 13.1: High level overview of the multi-output optimization layer. For a given list of views, we consolidate the views into view groups (Algorithm 13.2). For each group, we then compute a variable order of the join variables (Algorithm 13.3), and then register the views in the group and their aggregates to the variable order (Algorithms 13.5 and 13.6). The resulting register defines the multi-output optimization plan for the view group.

```
groupViews (View List V_list)
G_list = [ ];   V_grouped = ∅;   R_queue = { origin(V) | V ∈ V_list ∧ incoming(V) = ∅ };
while (R_queue ≠ ∅) {
  G = ∅;   R = R_queue.pop();
  foreach V ∈ V_list where V ∉ V_grouped ∧ origin(V) = R ∧ incoming(V) ⊆ V_grouped {
    G ∪= {V};   V_grouped ∪= {V};
    if (dest(V) ≠ ∅ ∧ dest(V) ∉ R_queue) { R_queue.push(dest(V)); }
  }
  G_list = G_list ++ G;
}
return G_list;
```

Algorithm 13.2: Pseudocode for the grouping of views. The input is a list of views, and the output a list of view groups. For a given view $V_{S \to T}$, let $\mathsf{incoming}(V)$ denote the set of incoming views to V, where $\mathsf{origin}(V) = S$ and $\mathsf{dest}(V) = T$ encode the direction of V.

same time. This means that all incoming views need to be computed before we evaluate the views in the group. Algorithm 13.2 presents the pseudocode for the procedure.

The algorithm takes as input a list V_{list} of directional views, which contains all merged views from Algorithm 12.2. We assume that the views are given in the order they were defined. In particular, any outgoing view $V_{R \to T}$ occurs after all intermediate views $V^i_{R \to T}$ in the body of $V_{R \to T}$. In a nutshell, the algorithm creates a topological order of the views in V_{list}, and groups consecutive views in the order that originate from the same relation.

For a view V with direction $S \to T$, let $\mathsf{origin}(V) = S$ and $\mathsf{dest}(V) = T$. In addition, we define $\mathsf{incoming}(V)$ to be the set of all incoming views in the join body of V.

The algorithm initializes: (1) an empty list G_{list} of view groups, (2) an empty set V_{grouped} of views, which keeps all views that have been assigned to a group, and (3) a queue R_{queue} of join tree nodes, which initially contains all leaf nodes of the join tree. The set of leafs in a join tree is equivalent to the set of origin nodes of all views that have no incoming views.

The algorithm then continuously removes the first relation R from the queue R_{queue}, and constructs a view group G. This group G contains all views V in V_{list} that are not already assigned to a group (i.e., they are not in V_{grouped}) and satisfy two conditions: (1) the view V originates from R, and (2) the set of all incoming views has already been assigned to a group (i.e. they are already in V_{grouped}). For each view V in G, the algorithm then adds this view to view group G and the set V_{grouped}. If the view has a destination, then we add the destination to R_{queue}. The group G is then added to the list of groups G_{list}, and the algorithm continues with the next element in R_{queue}. Once there are no more relations in R_{queue}, the algorithm returns the list G_{list} of view groups.

Example 13.1. Figure 12.1(left) shows queries Q_1, Q_2, and Q_3 and their directional views from Examples 12.2, 12.4, and 12.8 along the edges of our Favorita join tree. Figure 12.1(right) shows a dependency graph for the grouping of the directional views.

For each relation in the join tree, there are at most two groups that are computed over this relation, and there is only one group (Group 6) which is computed over the Sales relation, the largest relation in the dataset.

In the following examples, we focus on the computation of the three outputs in Group 6. For simplicity, we assume that each incoming and outgoing view only computes a single aggregate. The view group thus contains the following views:

$$\begin{aligned}
Q_1(\texttt{SUM(}f(\text{units}) \cdot \alpha_I \cdot \alpha_H \cdot \alpha_T\texttt{)}) &\leftarrow S(\text{item}, \text{date}, \text{store}, \text{units}, \text{promo}),\\
&\quad V_{T\to S}(\text{date}, \text{store}, \alpha_T), V_{I\to S}(\text{item}, \alpha_I), V_{H\to S}(\text{date}, \alpha_H)\\
Q_3(\text{store}, \texttt{SUM(}g(\text{item}) \cdot h(\text{date}, \text{family}) \cdot \alpha_H \cdot \alpha_T \cdot \alpha_I'\texttt{)}) &\leftarrow S(\text{item}, \text{date}, \text{store}, \text{units}, \text{promo}),\\
&\quad V_{H\to S}(\text{date}, \alpha_H), V_{T\to S}(\text{date}, \text{store}, \alpha_T), V_{I\to S}^{'}(\text{item}, \text{family}, \alpha_I')\\
V_{S\to I}(\text{item}, \texttt{SUM(}f(\text{units}) \cdot g(\text{item}) \cdot \alpha_H \cdot \alpha_T\texttt{)}) &\leftarrow S(\text{item}, \text{date}, \text{store}, \text{units}, \text{promo}),\\
&\quad V_{T\to S}(\text{date}, \text{store}, \alpha_T), V_{H\to S}(\text{date}, \alpha_H)
\end{aligned}$$

To ease the notation, we use V_H, V_T, V_I, and V_I' in the following examples to denote the views $V_{H\to S}$, $V_{T\to S}$, $V_{I\to S}$, and respectively $V_{I\to S}'$. We also use V_S to denote $V_{S\to I}$. □

LMFAO sees the incoming and outgoing views as functions that, for a given tuple over the group-by variables, look up the corresponding aggregate value. The aggregates to compute are also functions, in particular sums of products of functions that are UDAFs or lookups into incoming views.

Example 13.2. For instance, the aggregate of Q_1 is the product $f(\text{units}) \cdot \alpha_I \cdot \alpha_H \cdot \alpha_T$, where the last three components are provided by lookups in incoming views: $\alpha_I = V_I(i)$ for the aggregate α_I in the view V_I, where the group-by variable item is set to i; similarly for $\alpha_H = V_H(d)$ and $\alpha_T = V_T(d, s)$. □

variableOrder (View Group G)

let : $G = [V_1(\omega_{V_1}, \texttt{SUM(}\alpha_1\texttt{)}), \ldots, V_g(\omega_{V_g}, \texttt{SUM(}\alpha_g\texttt{)})]$ and $R = \mathsf{origin}(V_1)$
let : $\mathsf{incoming}(G) = \bigcup_{V \in G} \mathsf{incoming}(V)$ and $\omega_{\text{join}} = \{$ join variables in R $\}$

$\Lambda = [(\emptyset, \emptyset)]; \qquad \omega_\Lambda = \emptyset;$

$L = [\ X_j \in \omega_{\text{join}} \mid \forall i < j, |\mathsf{Dom}(L[i])| \leq |\mathsf{Dom}(L[j])|\ ]$ /* ω_{join} ordered by domain size */

foreach $X_\lambda \in L$ **do** {
  $\omega_\Lambda \ \cup\!= X_\lambda; \qquad \omega_N = \emptyset;$
  foreach $V(\omega_V, \texttt{SUM(}\alpha\texttt{)}) \in \mathsf{incoming}(G)$ **do** {
    if $((\omega_V \cap \omega_{\text{join}}) \subseteq \omega_\Lambda)$ { $\omega_\lambda \ \cup\!= \omega_V;$ }
  }
  $\Lambda = \Lambda \mathbin{+\!\!+} (X_\lambda, \omega_\lambda);$ /* Append pair of variable X_j and coverage set ω_λ */
}
return Λ;

Algorithm 13.3: Pseudocode that computes an order of the join variables in view group G. Each index λ in the order consists of a pair of join variable X_λ and coverage set ω_λ. $\mathsf{Dom}(X_i)$ denotes the active domain of variable X_i.

13.2 Join Variable Order

A view group is a computational unit in LMFAO. All views within a group are computed in one scan over their common relation. The scan uses a total order on the join variables of the relation and sees the relation logically as a (partial) trie, grouped by the first join variable and so on until the last join variable. The scan also sees the relations over the join variables in incoming views as tries that use the same variable order. The leaves of the tries are relations over the remaining non-join variables.

LMFAO sees variable order Λ as a list of nodes, where each node describes a level λ in the trie that it defines. The node at level λ is a pair $(X_\lambda, \omega_\lambda)$, where X_λ is a join variable, and ω_λ is a set of variables. The set ω_λ contains the set of join variables that are bound at this level in the order, as well as any non-join variables from the relation or incoming views whose join variables are bound. We call this set the "coverage set" of level λ.

Algorithm 13.3 presents how LMFAO computes the variable order Λ. It takes as input a view group G, which is a list of views. Let $\mathsf{incoming}(G)$ be the union of all incoming views for any view V in G, and ω_{join} be the set of join variables in the underlying relation R. The algorithm first initializes the variable order Λ with an "empty node", which is used in the following layers for the registration of views without group-by variables and of UDAFs that are constant functions. In addition, the algorithm initializes an empty set ω_Λ, which is used to store all join variables that have been added to Λ.

The algorithm then computes an ordered list L of the join variables ω_{join}. To avoid exploring all possible permutations of the join variables, LMFAO uses the following approximation. We first compute the size of the active domain $\mathsf{Dom}(X_i)$ for each join variable X_i,

i.e., the number of its distinct values. We choose the order that is the increasing order in the domain sizes of these variables.

For each join variable X_λ in L, the algorithm then computes the coverage set ω_λ and adds the pair $(X_\lambda, \omega_\lambda)$ to the variable order Λ. The coverage set ω_λ is composed of the group-by variables ω_V of all incoming views V for which the join variables in ω_V are covered by ω_λ, i.e., the set of all join variables that have already been added to Λ.

The variable order defines the join plan for the multi-output optimization. At each level λ in the variable order, we enumerate possible bindings for X_λ by looping over the result of a multi-way, unary join over the relation R and incoming views that contain X_λ, where the relation and views are selected on the bindings chosen for the join variables X_l at all levels $l < \lambda$. The execution plan thus uses a multi-way, variable-at-a-time join plan that is inspired by worst-case optimal join algorithms [133, 97] and factorized databases [102, 101].

We enumerate bindings for a non-join variable X_j by looping over the relation or incoming view whose schema contains X_j. For an incoming view V, we can enumerate the bindings for the non-join variables in the view at any level in the order where all join variables of the view are bound. Ideally, this enumeration is done at the highest possible level in the order to minimize the number of operations that are performed. We will see, however, that it may be necessary to defer the enumeration to a lower level in the order due to dependencies in the aggregate computation. For the relation, the join variables are always bound by the lowest level in the order, and thus we can only enumerate bindings for its non-join variables at the lowest level in the order.

This join strategy solves the first challenge for multi-output optimization: the joining of different incoming views, and in particular views from the same direction. This is possible because views from the same direction are computed over the same join, and thus their projection on the join variables must be the same.

Example 13.3. The group of views in Example 13.1 are computed over the relation S, where item, date, and store are join variables. Figure 13.1 (left) shows the order of the join variables we use to compute all views in the group.

The variable order defines an execution plan which first enumerates possible bindings $i \in \mathsf{Dom}(\text{item})$ for the join variable item from the result of the unary, multi-way join of relation S and the two incoming views V_I and V_I':

$$\pi_{\text{item}}(S \bowtie_{\text{item}} V_I \bowtie_{\text{item}} V_I')$$

The coverage set is $\omega_{\text{item}} = \{\text{item}, \text{family}\}$, because, given a binding for item, we can enumerate all possible bindings for family by looping over view V_I'.

The next variable in the order is date. The plan enumerates bindings $d \in \mathsf{Dom}(\text{date})$ for date from the unary join over the incoming views V_T and V_H and the partition of S that

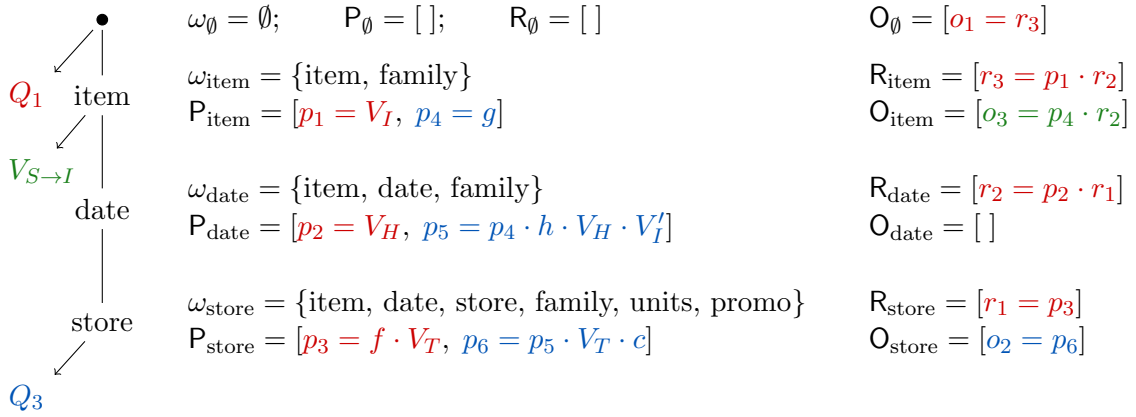


Figure 13.1: Annotated order of the join variables to compute Q_1, Q_3, and $V_{S\to I}$ from Example 13.1. Each level λ is annotated with the coverage set ω_λ, a list of products $\mathsf{P}[\lambda]$, running sums $\mathsf{R}[\lambda]$, and outgoing products $\mathsf{O}[\lambda]$, which contain products of functions that are computed at this level.

has value i for item:

$$\pi_{\text{date}}(\sigma_{\text{item}=i} S \bowtie_{\text{date}} V_H \bowtie_{\text{date}} V_T)$$

The coverage set for this node is $\omega_{\text{item}} = \{\text{item, date, family}\}$.

The final variable in the order is store. We enumerate bindings $s \in \mathsf{Dom}(\text{store})$ for store form the join:

$$\pi_{\text{store}}(\sigma_{\text{item}=i \wedge \text{date}=d} S \bowtie_{\text{store}} \sigma_{\text{date}=d} V_T)$$

where the coverage set $\omega_{\text{store}} = \{\text{item, date, store, family, units, promo}\}$, which includes all join and non-join variables that occur the multi-output plan.

We can enumerate the bindings for the non-join variable family at any level in the variable order after we fixed a binding for item. In contrast, we can only enumerate bindings for the non-join variables in S after we have fixed bindings for item, date, and store. □

13.3 View and Aggregate Registration

We now consider the second challenge for the multi-output optimization: computing the multiple outputs with a minimal number of operations.

We distinguish between two types of operations for the multi-output evaluation of view groups: (1) computing aggregates for intermediate views, which are products of functions, and (2) outputting tuples to outgoing views, where the tuples are over the group-by variables of the view and aggregates that are summations over products computed for (1).

A naïve execution strategy would compute the aggregates and output tuples to the views at the lowest level in the order of the join variables. This means that the computation is repeated for each possible binding over all join variables.

LMFAO instead factorizes the computation by decomposing a product of functions into

partial products, which can be evaluated at different levels in the variable order. Consider a partial product P which is evaluated at level λ in the order. The computation of P is then only repeated for each binding of the join variables at λ and any level $l \leq \lambda$ in the order, but not the bindings for join variables at the remaining levels. A function in the product can be evaluated at level λ in the order, if (1) all variables of the function are covered by ω_λ, and (2) the function does not depend on any other function that cannot evaluated at λ. Two functions depend on each other if they have non-join variables in the same relation or view. Dependent functions need to be evaluated together in loops over the bindings of the non-join variables. The reason why the dependency is defined only for non-join variables is that the join variables are bound by the nested unary joins over the variable order.

In addition, LMFAO outputs tuples to views at the highest possible level in the variable order, in order to minimize the number of times we update outgoing views. The tuples for outgoing view V can be constructed at level λ in the variable order if (1) ω_λ covers all group-by variables of V, and (2) there exists no partial product for this view that depends on the group-by variables of V and is evaluated at a lower node in the variable order.

Example 13.4. For query Q_1 from Example 13.1, we need to compute the product $f(\text{units}) \cdot \alpha_I \cdot \alpha_H \cdot \alpha_T$, where $\alpha_I = V_I(\text{item})$, $\alpha_H = V_H(\text{date})$, and $\alpha_T = V_T(\text{date}, \text{store})$ look up the aggregate in V_I, V_H and respectively V_T. Consider the order of the join variables item, date, and store from Figure 13.1.

A naïve evaluation would compute the product for inside the nested loops over bindings for each join variable item, date, and store, and the non-join variables in S:

$$
\begin{array}{l}
\textbf{foreach}\ \ i \in \pi_{\text{item}}(S \bowtie_{\text{item}} V_I \bowtie_{\text{item}} V_I')\ \ \textbf{do} \\
\quad \textbf{foreach}\ \ d \in \pi_{\text{date}}(\sigma_{\text{item}=i} S \bowtie_{\text{date}} V_H \bowtie_{\text{date}} V_T)\ \ \textbf{do} \\
\quad\quad \textbf{foreach}\ \ s \in \pi_{\text{store}}(\sigma_{\text{item}=i \wedge \text{date}=d} S \bowtie_{\text{store}} \sigma_{\text{date}=d} V_T)\ \ \textbf{do} \\
\quad\quad\quad \textbf{foreach}\ \ (u,p) \in \pi_{\text{units,promo}}(\sigma_{\text{item}=i \wedge \text{date}=d \wedge \text{store}=s} S)\ \ \textbf{do} \\
\quad\quad\quad\quad Q_1\ \mathrel{+}= f(u) \cdot V_I(i) \cdot V_H(d) \cdot V_T(d,s);
\end{array}
$$

Instead we can factorize the computation of the aggregate. A factorized evaluation looks up into $V_I(\text{item})$ the level of item, which means that we look up only once for each binding of item. Similarly, we can look up into $V_H(\text{date})$ and $V_T(\text{date}, \text{store})$ at level of date and respectively store:

$$
\begin{array}{l}
\textbf{foreach}\ \ i \in \pi_{\text{item}}(S \bowtie_{\text{item}} V_I \bowtie_{\text{item}} V_I')\ \ \textbf{do} \\
\quad p_1 = V_I(i); \\
\quad \textbf{foreach}\ \ d \in \pi_{\text{date}}(\sigma_{\text{item}=i} S \bowtie_{\text{date}} V_H \bowtie_{\text{date}} V_T)\ \ \textbf{do} \\
\quad\quad p_2 = V_H(d); \\
\quad\quad \textbf{foreach}\ \ s \in \pi_{\text{store}}(\sigma_{\text{item}=i \wedge \text{date}=d} S \bowtie_{\text{store}} \sigma_{\text{date}=d} V_T)\ \ \textbf{do} \\
\quad\quad\quad p_3 = V_T(d,s); \\
\quad\quad\quad \textbf{foreach}\ \ (u,p) \in \pi_{\text{units,promo}}(\sigma_{\text{item}=i \wedge \text{date}=d \wedge \text{store}=s} S)\ \ \textbf{do} \\
\quad\quad\quad\quad Q_1 \mathrel{+}= f(\text{units}) \cdot p_1 \cdot p_2 \cdot p_3;
\end{array}
$$

We can also minimize the number of times we update the output for Q_1. Since Q_1 does not have any group-by variables, the output is a scalar that captures the aggregate of the query. Instead of outputting partial results for each combination of item, date, and store bindings, we can accumulate the aggregate as running sums over the join variable order to output the result only once:

$$
\begin{array}{l}
r_1 = 0; \\
\textbf{foreach}\ \ i \in \pi_{\text{item}}(S \bowtie_{\text{item}} V_I \bowtie_{\text{item}} V_I')\ \ \textbf{do} \\
\quad p_1 = V_I(i); \quad r_2 = 0; \\
\quad \textbf{foreach}\ \ d \in \pi_{\text{date}}(\sigma_{\text{item}=i} S \bowtie_{\text{date}} V_H \bowtie_{\text{date}} V_T)\ \ \textbf{do} \\
\quad\quad p_2 = V_H(d); \quad r_3 = 0; \\
\quad\quad \textbf{foreach}\ \ s \in \pi_{\text{store}}(\sigma_{\text{item}=i \wedge \text{date}=d} S \bowtie_{\text{store}} \sigma_{\text{date}=d} V_T)\ \ \textbf{do} \\
\quad\quad\quad p_3 = V_T(d,s); \quad r_4 = 0; \\
\quad\quad\quad \textbf{foreach}\ \ (u,p) \in \pi_{\text{units,promo}}(\sigma_{\text{item}=i \wedge \text{date}=d \wedge \text{store}=s} S)\ \ \textbf{do} \\
\quad\quad\quad\quad r_4 \mathrel{+}= f(\text{units}); \\
\quad\quad\quad r_3 \mathrel{+}= p_3 \cdot r_4; \\
\quad\quad r_2 \mathrel{+}= p_2 \cdot r_3; \\
\quad r_1 \mathrel{+}= p_1 \cdot r_2; \\
Q_1 = r_1;
\end{array}
$$

This evaluation strategy minimize the number of function calls and the number of times we update the output over the given variable order.

The query Q_3 computes the product $h(\text{date}, \text{family}) \cdot g(\text{item}) \cdot \alpha_H \cdot \alpha_T \cdot \alpha_I'$, where $\alpha_I' = V_I'(\text{item}, \text{family})$ looks up the aggregate in V_I'. The function $g(\text{item})$ can be evaluated at the level of item. Given a binding i for item, we could also enumerate all matching bindings for family from $\sigma_{\text{item}=i} V_I'$ at the level of item. Note however that the functions $h(\text{date}, \text{family})$ and $V_I'(\text{item}, \text{family})$ depend on each other, because they share the non-join variable family. Their product needs to be evaluated together at the level of date, which is the lowest level in the order that covers all variables in the two functions. Therefore, we defer the enumeration

dependentFunctions (View V, Variable Set ω_{in})

let : $V(\omega_V, \texttt{SUM(}P\texttt{)}) \leftarrow \mathsf{body}$ where $P = [f]_{f \in [P]}$ and $\omega_{\text{join}} = \{$ join variables in $\mathsf{body}\ \}$
let : $dep(\omega) = \{S(\omega_S) \in \mathsf{body} \mid (\omega \setminus \omega_{\text{join}}) \cap \omega_S \neq \emptyset\}$

$F_{\text{dep}} = [\,]; \qquad S_{\text{dep}} = dep(\omega_{\text{in}});$
do {
  $c = |F_{\text{dep}}|;$
  foreach $f(\omega_f) \in P$ where $f \notin F_{\text{dep}}$ {
    if $(dep(\omega_f) \cap S_{\text{dep}} \neq \emptyset)$ { $F_{\text{dep}} = F_{\text{dep}} +\!\!+ f; \qquad S_{\text{dep}} \cup\!= dep(\omega_f);$ }
  }
} **while** $(c < |F_{\text{dep}}|);$
return $(F_{\text{dep}}, \bigcup_{S \in S_{\text{dep}}} \omega_S);$

Algorithm 13.4: Procedure that, for a given variable set ω_{in}, finds the functions in a list of functions that share non-join variables ω_{in}. The list of functions defines product aggregate of some intermediate view of the form (12.5).

of family bindings to the level of date. Query Q_3 has variable store as a group-by variable, and thus we need to update the output for Q_3 at the lowest level in the order. □

To support the factorized evaluation of multi-output plans, LMFAO constructs a register over the join variable order, which stores, for each level λ in the order, lists of partial aggregates that can be evaluated at λ. The register also keeps track of a list of outgoing views for which we can output tuples at λ.

We next present the pseudocode for the systematic registration of outgoing views and their aggregates to an order of the join variables. We first present and exemplify the pseudocode that identifies a list of functions in a product that depend on a given variable set. Then we show how outgoing views of the form (12.6) are registered to the variable order, and finally how we decompose the computation of the aggregates for intermediate views of the form (12.5) over the order of the join variables.

13.3.1 Dependent functions

We present a procedure that, given a set of variables ω_{in} and a product P of functions, identifies the functions in P that depend on non-join variables in ω_{in}. The challenge is to identify chains of dependencies: a function $f(\omega_f)$ that does not depend on ω_{in} can become dependent through another function g that depends on both ω_{in} and ω_f.

Algorithm 13.4 presents the pseudocode of this procedure. It takes as input an intermediate view V of the form (12.5) and a set of variables ω_{in}. Recall that the aggregate of V is a product P of functions over the join of a relation and incoming views. For this algorithm, we do not distinguish between relations and views, and thus see the join body simply as a join of relations, where each relation S has schema ω_S. If S corresponds to an incoming view, the schema ω_S is the set of group-by variables of the view. A function $f(\omega_f)$ in P

depends on ω_{in}, if the intersection $\omega_f \cap \omega_{\text{in}}$ contains any non-join variables in the schemas ω_S of all relations S in the join body of V.

Let ω_{join} denote the join variables in the body of V. For a given variable set ω, the function $dep(\omega)$ returns the set of relations in the body of V whose schema intersects with the non-join variables in ω.

The algorithm initializes an empty list F_{dep} of functions, and a set S_{dep} of relations in the body of V whose schema intersects with the non-join variables in ω_{in}.

The algorithm continuously iterates over each function $f(\omega_f)$ in product P that has not already been added to the list F_{dep} and checks if the set $dep(\omega_f)$ of dependent relations intersects with S_{dep}. If so, the function is added to the list F_{dep} of dependent functions, and the set S_{dep} is extended with the dependent relations in $dep(\omega_f)$. If additional functions are added to F_{dep}, the process is repeated to check if other functions in P depend on the newly added functions. Otherwise, the algorithm outputs a pair of the function list F_{dep} and the union of the schema of the relations in S_{dep}.

Example 13.5. We have seen in Example 13.4 that for Q_3 the functions $h(\text{date}, \text{family})$ and $V_I'(\text{item}, \text{family})$ depend on each other, which means that they need to be evaluated together in the loop over V_I' that provides bindings for family. For any input variable set ω_{in} which contains the variable family, the algorithm would thus return a list F_{dep} of functions, which contains both h and the look-up function for V_I', as well as a set of variables that contains the variables {date, item, family}.

Similarly, in Q_1 the function $f(\text{units})$ depends on the non-join variables of S. For any input variable set ω_{in} which contains any non-join variable in S, the algorithm would return the function f and a variable set which contains all variables in S.

Consider a function $p(\text{units}, \text{family})$. This function depends on non-join variables in both V_I' and S, and thus needs to be computed in nested loops over V_I' and S. For any input variable set ω_{in} which contains any non-join variable in S or V_I' the algorithm would return a list of functions which contains p and the look up function V_I', as well as a variable set that contains all variables in V_I' and S. □

We next present the procedure that registers outgoing views to the order of the join variables. This procedure uses Algorithm 13.4 to determine the level in the order where the view can be registered.

13.3.2 View registration

Algorithm 13.5 constructs a view register V for a view group G, which stores a list of views for each level λ in a variable order Λ. The views in $\mathsf{V}[\lambda]$ correspond to the outgoing views in G for which we construct and output tuples at level λ. The algorithm populates the register one-at-a-time, for each outgoing view V in G. The view V and variable order Λ are provided as input to the algorithm. Recall that outgoing view V is of the form (12.6). The

registerView (View V, Variable Order Λ)

let : $V(\omega_V, P_{\text{list}}) \leftarrow V_1(\omega_V, P_1), \ldots, V_s(\omega_V, P_s)$ where $P_{\text{list}} = [P_1, \ldots, P_s]$;
let : $\Lambda = [\ N_1, \ldots, N_{|\Lambda|}\ ]$ and $\omega_{\text{join}} = \bigcup_{N \in \Lambda} X_N$;

$\omega_{\text{dep}} = \omega_V$;
foreach $j \in [s]$ **do**{
  $(F_j, \omega_j) =$ **dependentFunctions**$(V_j,\ \omega_V \setminus \omega_{\text{join}})$;
  $\omega_{\text{dep}} \cup= \omega_j$;
}
foreach $\lambda \in [|\Lambda|]$ where $(X_\lambda, \omega_\lambda) = \Lambda[\lambda]$ **do** {
  if $(\omega_{\text{dep}} \subseteq \omega_\lambda)$ { $\lambda_{\text{out}} = \lambda$; **break**; }
}
foreach $j \in [s]$ **do** { $\alpha_j =$ **registerAggregate**$(V_j, \Lambda, 1, \lambda_{\text{out}}, [\ ], \omega_{\text{dep}} \setminus \omega_{\text{join}})$; }
$\mathsf{V}[\lambda_{\text{out}}] = \mathsf{V}[\lambda_{\text{out}}] +\!\!+ V(\omega_V, \alpha_1 + \ldots + \alpha_s)$;

Algorithm 13.5: Pseudocode for the registration of an outgoing view V of the form (12.6) to the variable order Λ.

join of V is over intermediate views of the form (12.5), and the aggregate is a summation over the aggregates from intermediate views.

To identify the highest level λ where we can register the outgoing view, we first construct a set ω_{dep} of variables, which contains (1) the set ω_V of group-by variables of V, and (2) the union of all "dependent" variables in any product aggregate for the intermediate views $(V_j)_{j \in [s]}$ in the join body of V. The set of dependent variables for a product is provided by Algorithm 13.4, where the input is the intermediate view V_j and its set of group-by variables. Once the set ω_{dep} is computed, the algorithm then finds the first level λ in the variable order for which ω_{dep} is a subset of the coverage set ω_λ. This level λ is then denoted as the "output level" λ_{out}, and denotes the level where the view is registered in the order.

The algorithm then calls the **registerAggregate** function from Algorithm 13.6 for each intermediate view $(V_j)_{j \in [s]}$. For a given view V_j, the function registers the product aggregate of V_j to the aggregate register over the variable order and returns a function α_j which maps to the aggregate in the register that represents the product aggregate of V_j. We thus define the outgoing view V as a view with group-by variables ω_V and (possibly multiple) aggregates which are summations over the registered aggregates in the aggregate register of the variable order. This view V is then registered to the list $\mathsf{V}[\lambda_{\text{out}}]$ of outgoing views for the level λ_{out} in the variable order.

Example 13.6. For the three outputs Q_1, Q_3, and V_S in the group from Example 13.1, we register Q_1 to the empty node at the top of the order (as it does not have any group-by variables), Q_3 at the level of store, and V_S at the level of item. The registration is shown in Figure 13.1(left) by the outgoing edges on the left of variable order. □

If an outgoing view has group-by variables that are non-join variables, we need to construct and output tuples to the view inside nested-loops over the relation or incoming

registerAggregate (View V, Order Λ, Level λ, Level λ_{out}, List α_{prev}, Set ω_{dep})

let : $V(\omega_V, \texttt{SUM(}P\texttt{)})$ where $P = [f_1(\omega_{f_1}), \ldots, f_p(\omega_{f_p})]$ and $R = \mathsf{origin}(V)$;
let : $\Lambda = [\ (X_{N_1}, \omega_{N_1}), \ldots, (X_{N_\ell}, \omega_{N_\ell})\ ]$ and $\omega_\Lambda = \bigcup_{(X_\lambda, \omega_\lambda) \in \Lambda} X_\lambda$;
let : $\mathsf{register}(L, P) =$ **if** $(\exists i : L[i] = P)$ **return** $L[i]$ **else** $\{L = L +\!\!+ P;$ **return** $L[|L|]\}$;
let : $\mathsf{addCount}(P, \omega) =$ **return** $(\omega_{\text{dep}} \cap \omega = \emptyset \wedge \omega_f \cap \omega = \emptyset, \forall f(\omega_f) \in P)$;

$(X_\lambda, \omega_\lambda) = \Lambda[\lambda]$; **if** $(\lambda \leq \lambda_{\text{out}})$ $\alpha_\lambda = \alpha_{\text{prev}}$ **else** $\alpha_\lambda = [\ ]$; $\alpha_{\text{dep}} = [\ ]$;

foreach $f \in P$ **do** {
  $(S, \omega_S) =$ **dependentFunctions**$(V, \omega_f \setminus \omega_\Lambda)$;
  if $(\omega_S \subseteq \omega_\lambda)$ {
    if $(\omega_S \cap \omega_{\text{dep}} \neq \emptyset)$ { $\alpha_{\text{dep}} = \alpha_{\text{dep}} +\!\!+ S$; } **else** { $\alpha_\lambda = \alpha_\lambda +\!\!+ S$; }
    $P = [\ f \in P \mid f \notin S\ ]$;
  }
}

if $(\lambda + 1 > |\Lambda| \wedge \mathsf{addCount}(\alpha_\lambda, \omega_R \setminus \omega_\Lambda))$ $\alpha_\lambda = \alpha_\lambda +\!\!+ [c(\omega_R \setminus \omega_\Lambda) = 1]$;

$\alpha_{\mathsf{local}} = \mathsf{register}(\mathsf{P}[\lambda], \alpha_\lambda)$; $\alpha_{\mathsf{out}} = [\ ]$

if $(\lambda + 1 \leq |\Lambda|)$ $\alpha_{\mathsf{out}} =$ **registerAggregate**$(V, \Lambda, \lambda + 1, \lambda_{\text{out}}, [\alpha_{\text{local}}], \omega_{\text{dep}})$;

if $(\lambda > \lambda_{\text{out}})$ $\alpha_{\mathsf{out}} = \mathsf{register}(\mathsf{R}[\lambda], \alpha_{\mathsf{local}} +\!\!+ \alpha_{\mathsf{out}})$;

if $(\lambda = \lambda_{\text{out}})$ $\alpha_{\mathsf{out}} = \mathsf{register}(\mathsf{O}[\lambda], \alpha_{\text{dep}} +\!\!+ \alpha_{\mathsf{local}} +\!\!+ \alpha_{\mathsf{out}})$;

return α_{out};

Algorithm 13.6: Pseudocode for the registration of aggregates of views of the form (12.5) to variable order Λ. The aggregate is defined by a product P of functions, and decomposed into partial products. Each partial product is registered to a level λ in Λ.

view that supports the non-join variables.

13.3.3 Aggregate function registration

We next explain how we register the aggregates of an intermediate view V of the form (12.5) to the variable order. Recall that the aggregates of V are defined by a product P of functions.

The aggregate registration is done in two steps. First, we decompose the product P of functions into partial products which can be evaluated at different levels in the join variable order Λ. Each partial product is then registered to a list of products at the level where it is evaluated. The evaluation of these partial products may require additional loops that bind any non-join variables in the product. The second step then considers a list of partial products at a given level λ, and decomposes these products over the loops that bind non-join variables. The computation for the second step is very similar to the first. We therefore first focus on the registration to the join variable order, for which we present a pseudocode and motivate the problem with examples. Then we exemplify how the registered products can be decomposed over the loops for non-join variables. The resulting aggregate register

forms a template for the code generation step.

Algorithm 13.6 presents the pseudocode of the procedure which decomposes a product P of functions into partial products, where each partial product is registered to a list of products at one level λ in the variable order Λ. For each level λ, the aggregate register keeps three lists of products: (1) $\mathsf{P}[\lambda]$ is a list of partial products that can be evaluated at the level λ; (2) $\mathsf{R}[\lambda]$ is a list of running sums, where each running sum represents an accumulated product of the functions that are evaluated at any level $\lambda \leq l < |\Lambda|$; and (3) $\mathsf{O}[\lambda]$ is a list of "output" products. Each element in $\mathsf{O}[\lambda]$ is a product of (1) functions in P that depend on the group-by variables of the view V, (2) an element in $\mathsf{P}[\lambda]$, and (3) a running sum in $\mathsf{R}[\lambda + 1]$ of some product in $\mathsf{P}[\lambda]$ and a running sum in $\mathsf{R}[\lambda + 1]$. Each output product is equivalent to one product of functions of an intermediate view that is decomposed over the variable order.

The procedure is recursive and decomposes a given product over the variable order Λ one level at a time. The input to the algorithm is an intermediate view V of form (12.5), a variable order Λ, two integers λ and λ_{out}, a function list α_{prev}, and a set ω_{dep} of variables. The aggregate of view V defines a product P of functions. The integers λ and λ_{out} denote levels in Λ. In particular, λ denotes the current level of the algorithm in Λ, and λ_{out} is the "output level", i.e. the level where we output tuples to the outgoing view. The set ω_{dep} denotes the set of non-join variables that the outgoing view depends on. Both λ_{out} and ω_{dep} are provided by Algorithm 13.5. Finally, the list α_{prev} encodes a single function that is a look up into the aggregate register for the partial product that was registered at level $\lambda - 1$. If no partial product was registered any previous level, then the list is empty.

For each level λ, the algorithm identifies partial products in P, which can be evaluated at the level λ. We differentiate between two types of partial products that can be evaluated at λ: products of functions which (1) do not and (2) do share non-join variables with the group-by variables of the outgoing view. Products of the former are represented by α_λ, whereas the latter are kept in α_{dep}. Since functions in α_{dep} share non-join variables with the group-by variables in the outgoing view, they need to be evaluated in the same loops over the non-join variables which are used to construct the tuples of the outgoing view. If the current level $\lambda < \lambda_{\text{out}}$, then the α_λ is initialized to α_{prev}, otherwise it is empty. The product α_{dep} is always empty initially.

We next explain how we populate the products α_λ and α_{dep}. For each function $f(\omega_f)$ in the product P, we use Algorithm 13.4 to compute the list S of functions in P that dependent of the non-join variables in ω_f, as well as the set ω_S of dependent variables for the functions in S. We then check if the set ω_S is covered by ω_λ. If so, we can register the partial product defined by S at λ. If $\omega_S \cap \omega_{\text{dep}} \neq \emptyset$, we add the functions in S to α_{dep}, otherwise we add them to α_λ. The functions in S are then removed from product P to avoid that we register the same functions again at a later level in the order.

Once the products α_λ and α_{dep} are populated, we check if we need to extend the product

α_λ with a special function c. The intuition for this function is as follows. Let R denote the underlying relation of the view group. Contrary to the incoming views, R has not been pre-aggregated, which means that any product that is registered to the variable order needs to be computed over the all tuples in R. Any product that does not require a loop over the non-join variables in R, thus needs to be multiplied by the count of tuples in the fragment of R that is selected on the bindings for the join variables. (This count is implicitly defined by `SUM` in the aggregate expression of the view.) We use function c to denote this count; it is defined over the non-join variables in R and always returns 1. The function c is added to α_λ if the following conditions hold: (1) λ is the last level in the Λ, (2) the outgoing view does not depend on the non-join variables in R (otherwise we would enumerate the non-join variables in the output and do not require their count), and (3) there is no function in α_λ depends on the non-join variables in R.

We next proceed with the registration of the partial products to the lists $\mathsf{P}[\lambda]$, $\mathsf{R}[\lambda]$, and $\mathsf{O}[\lambda]$. We first register the product α_λ to the list $\mathsf{P}[\lambda]$. This is done with a registration function, which first checks if α_λ has already been added to $\mathsf{P}[\lambda]$. If so, it returns the reference to the previously defined product, and otherwise it appends α_λ to $\mathsf{P}[\lambda]$ and returns the reference to the new element in the list. LMFAO sees the returned reference as a look-up function α_{local} into the aggregate register.

After the registration of α_λ, we recurse to the next level in the order, where α_{prev} is now defined by α_{local}. The recursive call returns another look-up function α_{out} for an element in the aggregate register that is registered at level $\lambda + 1$.

If the current level $\lambda > \lambda_{\text{out}}$, then we output tuples to the outgoing views at a level above λ. Therefore, we register a running sum to $\mathsf{R}[\lambda]$ that accumulates the product of α_{local} and α_{out}, where α_{out} points to the running sum registered at the level $\lambda + 1$. We let α_{out} point to the newly registered running sum. The running sum represents the accumulated product of all functions in P that are evaluated at any level $l \geq \lambda$.

If the current level $\lambda = \lambda_{\text{out}}$, we output tuples to the outgoing view at this level. The aggregates in the tuples are defined by the product of α_{local}, α_{out}, and the dependent functions in α_{dep}. This product is added to the list $\mathsf{O}[\lambda]$.

Finally, we return α_{out}. Note that for any aggregate product that is decomposed over the variable order, the final α_{out} always maps to an element in the list $O[\lambda_{\text{out}}]$.

Example 13.7. Figure 13.1(right) shows the registration of the products for Q_1, Q_3, and $V_{S\to I}$ to the lists P, R, and O for each level in the variable order.

For Q_1 we decompose the product as described in Example 13.4: we register $V_I(\text{item})$ and $V_H(\text{date})$ to the list of products in P_{item} and respectively P_{date}. The remaining partial product $f(\text{units}) \cdot V_T(\text{date}, \text{store})$ is registered to store. The function f has a non-join variable from relation S, which implies (1) the product needs to be evaluated in a loop over the non-join variables in S, and (2) we do not multiply this product with the count function for S. For each partial product in Q_1, we register a running sum at the level of item, date,

and store, which is the product of the partial product computed at the same level, and the running sum computed at the level below. For instance, the running sum r_2 registered in R_{date} captures the product of p_2 and r_1, which in turn is the accumulated aggregate over the product p_3. The running sum r_2 thus represents the accumulated sum of the product $f(\text{units}) \cdot V_T(\text{date}, \text{store}) \cdot V_H(\text{date})$ over the possible date and store bindings, for a given item binding. We further register the outgoing product o_1 at the empty node, which shows that the output for Q_1 is equivalent to the running sum r_3 that accumulates the product for Q_1 over the entire variable order.

For Q_3, we register function $g(\text{item})$ at item, the partial product $h(\text{family}, \text{date}) \cdot V_H(\text{date}) \cdot V_i'(\text{item, family})$ at date, and function V_T(date,store) at store. The product registered to store does not have a function that has non-join variables from relation S, so we multiply the product with the function c(units,promo) $= 1$ to indicate that this function needs to be multiplied by the count of the non-join variables in S for the given bindings of the join variables. Since the levels λ_{item} and λ_{date} are above the output level λ_{store}, we also multiply each of these products with the product that was registered at the previous level, i.e. p_5 is multiplied by p_4, and p_6 is multiplied by p_5. We output Q_3 at the lowest level in the order, and thus we do not register any running sums. Instead, we register the output product o_2 in $\mathsf{O}_{\text{store}}$, which indicates that each output tuple for the binding of store is updated by the aggregate p_6.

For the outgoing view V_S, we would register $g(\text{item})$ at item, which is exactly the same as the partial product p_4 that was registered for Q_3. Therefore, we reuse p_2 and do not add an additional product to P_{item}. In addition, the outgoing view shares the computation for the levels of store and date with Q_1. This includes the partial products p_2 and p_3, but also the running sums r_1 and r_2. Finally, we register the output aggregate o_3 which is the product of p_4 and r_2 to O_{item}, which defines the aggregate for tuples in V_S. □

The aggregate register defines a template for the factorized evaluation of multiple outputs within a view group. At runtime, the products in $\mathsf{P}[\lambda]$ are evaluated after the loop over the bindings for join variable X_λ and *before* the loop over the bindings for the next join variable in the order. In contrast, the running sums in $\mathsf{R}[\lambda]$ and output aggregates in $\mathsf{O}[\lambda]$ are computed *after* the loop over the bindings of the next join variable. Figure 13.2 shows the execution plan that results from the aggregate register in Figure 13.1. The presented code exhibits additional optimizations for the registered products which we exemplify next.

13.3.4 Optimizing registered products

For each level λ in Λ, the products in $\mathsf{P}[\lambda]$ need to be computed inside nested loops over non-join variables. Similarly, the products in $\mathsf{O}[\lambda]$ need to be evaluated in the loops over the non-join variables of the corresponding view in $\mathsf{V}[\lambda]$. Instead of computing each product individually in its respective loop, LMFAO optimizes the computation of all products by fusing the loops and sharing computation across products. The optimization is composed

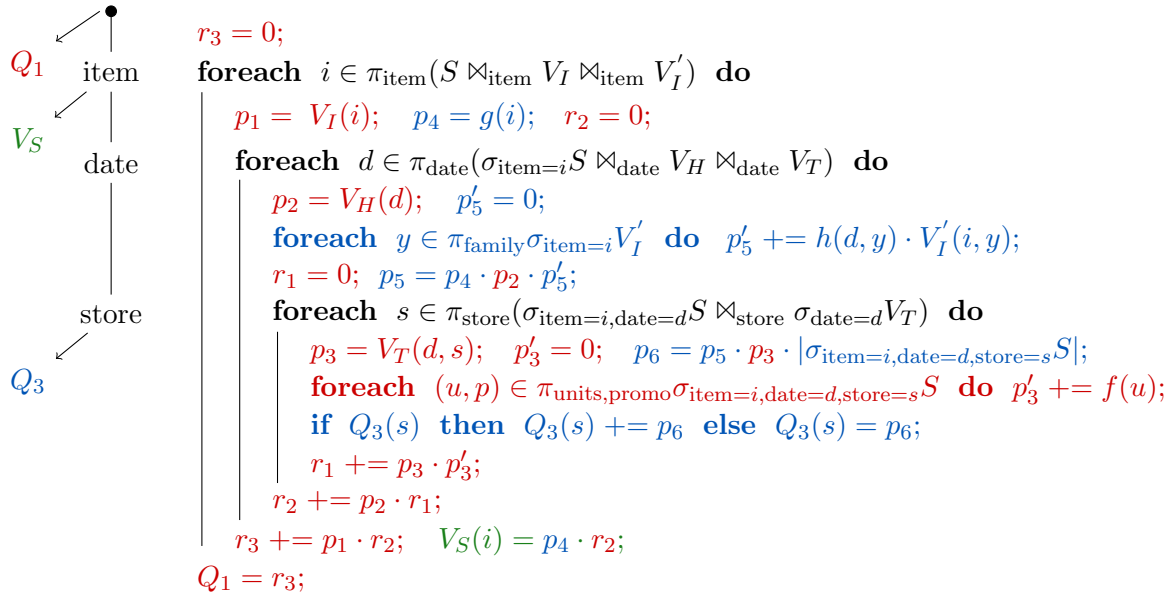


Figure 13.2: Multi-output execution plan to compute Q_1, Q_3, and V_S from Example 13.1 following the aggregate register from Figure 13.1.

of four steps: (1) for each list of products, we identify the loops that are required for this product, (2) we define an order on all loops that are required at level λ, (3) we fuse loops for products that have a common prefix in the loop order, (4) we decompose the computation of each product over its respective loop in the order.

Each of these four optimization steps is very similar to one of the algorithms presented above. We next highlight each optimization step, motivate them by examples, and explain how we can reuse the algorithms that decompose the aggregate over the join variable order to also optimize the computation over the loops for non-join variables.

We consider a list of products in $\mathsf{P}[\lambda]$, for some level λ in Λ. For a product $P \in \mathsf{P}[\lambda]$, let ω_P denote the union of the variable sets ω_f for each function $f(\omega_f)$ in P.

As for Algorithm 13.4, we see incoming views as relations over their group-by variables. The first optimization step identifies, for each product $P \in \mathsf{P}[\lambda]$, the set T_P of relations that support the non-join variables in ω_P. We compute the set T_P the *dep* function from Algorithm 13.4. Let T denote the *multiset* over the union of all relations sets T_P for each $P \in \mathsf{P}[\lambda]$, where the multiplicity for a relation in T represents the number of products in $\mathsf{P}[\lambda]$ that require a loop over this relation.

The second optimization step then constructs an order Γ of the relations in T, where the relations are ordered in decreasing order with respect to their multiplicities. The first relation in Γ is thus the relation that is looped over the most by all products in $\mathsf{P}[\lambda]$. We use this order for two optimizations. (1) We fuse nested loops over the non-join variables for different products in $\mathsf{P}[\lambda]$, for which the sets T_P share a common prefix in Γ. Fusing loops allows us to loop over the relation only once, and share computation across products.

(2) For each $P \in \mathsf{P}[\lambda]$, we decompose the computation of P over corresponding nested loops for the non-join variables in P, where the loops are ordered as defined by Γ. This can be done with a variant of Algorithm 12.2, where the loops are defined by relations over non-join variables instead of bindings for the join variables. In essence, we construct another aggregate register for the partial products in $\mathsf{P}[\lambda]$. The benefit of this optimization step is that we can share computation across the products in $\mathsf{P}[\lambda]$.

We next exemplify these optimization with a simple example, and then explain how these optimizations are represented in the code of Figure 13.2.

Example 13.8. Consider the following three products $p_1 = f(\text{units})$, $p_2 = f(\text{units}) \cdot g(\text{promo}, \text{family}) \cdot V_I'(\text{item}, \text{family})$, and $p_3 = h(\text{family}) \cdot V_I'(\text{item}, \text{family})$, where units and promo are non-join variables in relation S, and family is a non-join variable in incoming view V_I'. We assume that the products are registered to the last level in the variable order from Figure 13.1. We denote the bindings for join variables item, date, and store by i, d, and respectively s.

The *dep* function from Algorithm 13.4 would return the following relation sets for the three products: $T_{p_1} = \{S\}$, $T_{p_2} = \{S, V_I'\}$, and $T_{p_3} = \{V_I'\}$. Let $\Gamma = [S, V_I']$ denote the order of the union over the relations T_{p_1}, T_{p_2}, and T_{p_3}.

The naïve evaluation strategy computes each product $(p_j)_{j \in [3]}$ individually in (nested) loops over the relations in T_{p_j} following the order of the loops given by Γ:

$$
\begin{array}{l}
\textbf{foreach}\ \ (u,p) \in \pi_{(\text{units},\text{promo})} \sigma_{\text{item}=i \wedge \text{date}=d \wedge \text{store}=s} S\ \ \textbf{do} \\
\quad p_1 \mathrel{+}= f(u); \\
\\
\textbf{foreach}\ \ (u,p) \in \pi_{(\text{units},\text{promo})} \sigma_{\text{item}=i \wedge \text{date}=d \wedge \text{store}=s} S\ \ \textbf{do} \\
\quad \textbf{foreach}\ \ y \in \pi_{\text{family}} \sigma_{\text{item}=i} V_I'\ \ \textbf{do} \\
\quad\quad p_2 \mathrel{+}= f(u) \cdot g(p,y) \cdot V_I'(i,y) \\
\\
\textbf{foreach}\ \ y \in \pi_{\text{family}} \sigma_{\text{item}=i} V_I'\ \ \textbf{do} \\
\quad p_3 \mathrel{+}= h(y) \cdot V_I'(i,y)
\end{array}
$$

Since $T_{p_1} = \{S\}$ and $T_{p_2} = \{S, V_I'\}$ share the same prefix in Γ we can loop over S only once for both products. $T_{p_3} = \{V_I'\}$, however, does not share a prefix with the other two sets, and therefore we cannot fuse the loop for p_3. In addition, we can decompose the product p_2 over the loops for S and V_I'. The evaluation of function f is then registered to the loop over S. As a result, we can evaluate the function f only once for both products p_1

and p_2. The optimized execution of the three products is then given by:

$$\begin{aligned}
&\textbf{foreach } (u,p) \in \pi_{(\text{units,promo})}\sigma_{\text{item}=i\wedge\text{date}=d\wedge\text{store}=s}S \textbf{ do}\\
&\quad q = f(u); \quad p_1 \mathrel{+}= q;\\
&\quad \textbf{foreach } y \in \pi_{\text{family}}\sigma_{\text{item}=i}V_I' \textbf{ do}\\
&\qquad p_2 \mathrel{+}= q \cdot g(p,y) \cdot V_I'(i,y)\\
&\textbf{foreach } y \in \pi_{\text{family}}\sigma_{\text{item}=i}V_I' \textbf{ do}\\
&\quad p_3 \mathrel{+}= h(y) \cdot V_I'(i,y)
\end{aligned}$$

□

Example 13.9. Figure 13.2 depicts the evaluation plan for Q_1, Q_3, and V_S. The aggregates for the three outputs are evaluated at the different levels in the order, following their registration in the aggregate register from Figure 13.1.

For instance, the products p_1, p_1, and p_3 for Q_1 are registered at the level of item, date, and respectively store. Each product $P \in \mathsf{P}[\lambda]$ that requires a loop over non-join variables is decomposed into smaller products, such that only the functions in P that depend on the non-join variables are computed inside the loop. For instance, the product p_3' and p_5' are components of p_3 and respectively p_5 which are computed inside the loop, and then multiplied by the other functions in the product outside the loop.

This decomposition enables further sharing opportunities. For instance, we can share the look-up into V_H at the level of date for both products p_2 and p_5. We can also share the look-up into V_T between products p_3 and p_6.

Another optimization is presented for the count function c. This function avoids the loop over the non-join variables, and simply returns the size of the fragment of S with the given bindings for item, store, and date. This is possible, because LMFAO stores S as an array that is sorted by the join variables. Thus, this fragment is a contiguous range in the array, whose size can be provided right away without having to enumerate over it.

The running sums in $\mathsf{R}[\lambda]$ are initialized before the loop over bindings for the join variable at λ. Then, they are updated after the loop over bindings for the next join variable. For instance, r_1 is initialized before the loop over item, and then updated after the loop over date bindings.

We do not explicitly define local variables for output products in $\mathsf{O}[\lambda]$, but instead inline their computation with the update of the output. Since Q_1 has no group-by variables, its result is the scalar given by the running sum r_3: $Q_1 = r_3$. We insert tuples in Q_3 within the loop over stores and update the aggregate value for a given store if the same store occurs under different (item, date) pairs. V_S reuses the running sum r_2 computed for Q_1 and product p_4 computed for Q_3. The tuples for V_S are constructed in the order of the items enumerated in the outermost loop. □

Chapter 14

Code Generation

The code generation layer performs low-level code optimizations on the multi-output execution plan generated in the previous layer. The output of this layer is succinct and efficient C++ code which is specialized for the shared and parallel computation of many aggregates in a view group. The optimizations are inspired by recent work on code compilation for query evaluation [95, 74, 124, 123, 108, 130], and can be classified into four groups: (1) We choose specialized data structures that support the efficient computation of views and their aggregates within a view group ; (2) we synthesize loops to optimize the computation of aggregates that are computed in sequence; (3) we use compilation techniques to inline function calls and reduce the interpretation overhead of the plan; (4) we parallelize the computation of the multi-output plans for different view groups. We next highlight the optimizations performed in this optimization layer. The generated code closely follows the multi-output plan similar to that in Figure 13.2.

14.1 Data Structures

The database catalog and join tree provide a lot of statistics that LMFAO exploits to generate specific data structures to represent the relations and views. Given its size and schema, a relation is represented as a fixed size array of tuples that are represented using specialized C++ structs with the exact type for each variable.

We can also define specialized C++ structs that represent tuples in views. The data structure that represents the view depends on how the execution plan updates the view. If the plan outputs tuples to the view in sequence, then we represent the view as a vector, and any update appends tuples to this vector. If the view is not updated in sequence, we represent the view as a hash map instead, which supports efficient out-of-order updates.

Example 14.1. For the multi-output execution plan from Figure 13.2, the optimization chooses different data structures to represent the results of Q_3 and V_S. Since we iterate over distinct items and V_S has one tuple per item, we can store V_S as a vector where the tuple

```
aggregate[35] += aggregateV3[0];
        ...
aggregate[44] += aggregateV3[9];
aggregate[45] += aggregate[0]*aggregateV3[0];
        ...
aggregate[73] += aggregate[28]*aggregateV3[0];
aggregate[74] += aggregate[0]*aggregateV3[1];
aggregate[75] += aggregate[0]*aggregateV3[2];
aggregate[76] += aggregate[0]*aggregateV3[3];
```

(a) Aggregates are stored and accessed consecutively in fixed size array.

```
for (size_t i = 0; i < 10;++i)
  aggregate[35+i] += aggregateV3[i];
for (size_t i = 0; i < 29;++i)
  aggregate[45+i] += aggregate[i]*aggregateV3[0];
for (size_t i = 0; i < 3;++i)
  aggregate[74+i] += aggregate[0]*aggregateV3[i];
```

(b) Updates to consecutive aggregates are fused into tight loops.

Figure 14.1: Snippet of code generated by LMFAO that shows how aggregates are stored in fixed size array. The computation of the aggregates is synthesized into tight loops.

over each new item value is appended. In contrast, the plan may encounter the same store under different (item, date) pairs. The optimization therefore represents Q_3 as a hashmap to support efficient out-of-order updates. □

Another optimization concerns the representation of the aggregate computed in the multi-output optimization plan. The registration of aggregates to the variable order allows us to derive at compile time how many aggregates are computed in a group and the order in which they are accessed. LMFAO uses this information during the code generation to generate fixed size arrays that store all aggregates consecutively, in an order that allows for sequential reads and writes.

Example 14.2. Figure 14.1(a) presents a snippet of generated code that computes partial aggregates for the covar matrix. Each aggregate array is accessed sequentially, which improves the cache locality of the generated code. □

14.2 Loop Synthesis

The sequential access to the array of aggregates further allows us to compress long sequences of arithmetic operations over aggregates addressed in lockstep into tight loops. This optimization allows the compiler to vectorize the computation of the loop and reduces the amount of code to compile.

Example 14.3. Figure 14.1(a) presents a block of aggregate computations that is a common component for the execution of view groups. Within the block, there are sequences of computations that access the arrays in lockstep. LMFAO can efficiently synthesize the computation of these sequences into tight loops. Figure 14.1(b) shows how the aggregates in Figure 14.1(a) are computed after loop synthesis. □

14.3 Inlining Function Calls

LMFAO further optimizes the computation of the joins and aggregates by inlining function calls. For each join, the variable order gives the join variable and the views that are joined on. LMFAO uses this information to generate specialized code that computes these joins without dynamic casting and iterator function calls.

For the computation of aggregates, LMFAO knows which UDAFs are computed at compile time, and can thus inline these functions during code generation.

Example 14.4. For the learning of decision trees over Retailer, LMFAO computes the following product of indicator functions for a given tuple `t`:

$$p = \mathbf{1}_{(\texttt{t.avghh}<=52775)} \cdot \mathbf{1}_{(\texttt{t.area\_sq\_ft}>93580)} \cdot \mathbf{1}_{(\texttt{t.distance\_comp}<=5.36)}$$

In the generated C++ code, the product p is computed with the following code snippet, which inlines the three indicator functions as simple boolean conditions:

```
aggregate[96] = (t.avghhi <= 52775) * (t.area_sq_ft > 93580) * (t.distance_comp <= 5.36);
```

□

Some workloads require repeated computation of slightly different aggregates. To learn decision trees, for instance, we repeatedly compute the same set of aggregates, where the only difference is one additional threshold function per node. To avoid recompiling the entire generated code for each decision tree node, we generate dynamic functions in a separate C++ file with a few lines of code. This file can be recompiled efficiently and dynamically loaded into a running instance of LMFAO to recompute the aggregates.

14.4 Parallelization

LMFAO exploits both task and domain parallelism. First, LMFAO parallelizes the computation of multi-output plans for view groups that do not depend on each other. Second, LMFAO partitions the largest input relations and allocates a thread per partition to compute the multi-output plan on that partition.

Example 14.5. Recall the dependency graph of the view groups from Figure 12.1. LMFAO parallelizes the computation of groups 1, 2, 3, and 4. Once the computation of groups 1 and 2 finishes, LMFAO computes group 5, possibly in parallel with groups 3 and 4. □

Chapter 15

Discussion: Applications in LMFAO

In this chapter, we discuss how LMFAO computes and optimizes linear regression models, regression trees, and k-means clustering over databases.

For each model, we first assume that the training dataset is defined by a join query with n variables that is computed over database I with relations $R_1, \ldots, R_m$. Then, we exemplify the number of aggregates, views, and view groups that are computed for the Favorita dataset from Example 12.1.

15.1 Linear Regression

LMFAO learns a linear regression model as explained in Section 6.3: First, it computes a batch of aggregates that define the covar matrix, and then it learns the model parameters using batch gradient descent optimization with the Armijo line search condition and Barzilai-Borwein step size adjustment [24, 51], as presented in Algorithm 6.1.

In total, LMFAO computes $\binom{n+1}{2}$ aggregates queries, one for each pairwise combination of the variables in the feature extraction query. For the Favorita dataset, this translates into a batch of 140 aggregates.

Consider the two variables X_j and X_k in the feature extraction query. As explained in Section 6.3, the query that defines the corresponding entry in the covar matrix depends on the type of the variables. If X_j and X_k are continuous variables, then we compute a simple `SUM` aggregate query:

$$\mathsf{Covar}_{j,k}(\texttt{SUM}(X_j \cdot X_k)) \leftarrow R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}) \tag{15.1}$$

If X_j is continuous and X_k categorical, then we compute an aggregate query with X_k

as group-by variable:

$$\mathsf{Covar}_{j,k}(X_k, \texttt{SUM}(X_j)) \leftarrow R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}) \tag{15.2}$$

If both X_j and X_k are categorical, then we compute a count query with group-by variables X_j and X_k:

$$\mathsf{Covar}_{j,k}(X_j, X_k, \texttt{SUM(1)}) \leftarrow R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}) \tag{15.3}$$

For each query, LMFAO chooses one relation in the join tree as the root for the query. If there exists a relation whose schema contains both variables X_j and X_k then this relation becomes the root of the query. Otherwise, we choose as root the largest relation that contains either X_j or X_k.

The aggregate queries have very similar structure. For instance, for a continuous variable X_j, there are n queries in the batch that require the summation over X_j. LMFAO exploits this property to share computation across queries, via the decomposition of aggregate queries into views.

For instance, for the Favorita dataset, we require the computation of 125 views. Note that, even though we need to compute one view for each query and edge in the join tree, there are less views than aggregates in the batch. This shows that LMFAO can effectively share computation across aggregates in the batch. The views are then grouped into 9 view groups, which further share the scans over relations for views in the same view group. The output size of all aggregates is 900MB, whereas the materialized result of feature extraction query requires 7GB on disk. The learning task in LMFAO is therefore computed over an input that is orders of magnitude smaller than the result of the feature extraction query that defines training dataset for mainstream machine learning packages.

15.2 Regression Tree

LMFAO learns a regression tree in two steps. First, we compute percentiles for each continuous variable X_i that define the set $\mathcal{T}_i$ of possible threshold conditions. In practice, we compute between 20 and 100 conditions per continuous variable. Then, we learn the regression tree using the CART algorithm as explained in Chapter 8. The first step can be computed using one group-by count query for each continuous variable. In the following, we focus on the second step.

LMFAO computes a batch of aggregates for each node N in the regression tree, as shown in Section 8.2. Let $\mathcal{C}_N$ denote the set of threshold conditions along the path in the regression tree from the root to N. The aggregate batch contains one query of the following form for

each continuous variable X_i and threshold condition $t_i \in \mathcal{T}_i$:

$$\mathsf{RTree}(\mathtt{SUM}(\alpha), \mathtt{SUM}(Y \cdot \alpha), \mathtt{SUM}(Y^2 \cdot \alpha)) = R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}) \quad (15.4)$$
$$\text{where} \quad \alpha = \mathbf{1}_{X_i \text{ op } t_i} \cdot \prod_{[X_j \text{ op } t_j] \in \mathcal{C}_N} \mathbf{1}_{X \text{ op } t}$$

where α denotes a product of threshold conditions.

In addition, the batch contains one group-by query for each categorical variable X_i:

$$\mathsf{RTree}(X_i, \mathtt{SUM}(\alpha), \mathtt{SUM}(Y \cdot \alpha), \mathtt{SUM}(Y^2 \cdot \alpha)) = R_1(\omega_{R_1}), \ldots, R_m(\omega_{R_m}) \quad (15.5)$$
$$\text{where} \quad \alpha = \prod_{[X_j \text{ op } t_j] \in \mathcal{C}_N} \mathbf{1}_{X \text{ op } t}$$

The size of the batch thus depends on the type of the variables, as well as the chosen number of threshold conditions used for continuous variables.

The CART algorithm requires that the aggregate batch is recomputed for each node in the regression tree, where the only difference between the query batch for different nodes is the set $\mathcal{C}_N$ of threshold conditions. In order to avoid that LMFAO generates and compiles code for each node in the regression tree, we instead use dynamic functions to capture these conditions. These dynamic functions are generated in a separate file, which can be updated by the running instance of LMFAO, recompiled efficiently, and then dynamically loaded to recompute the aggregate batch. Note that all aggregates in the batch require the evaluation of the same dynamic functions. The optimization layers of LMFAO can efficiently share the evaluation of the dynamic functions across all queries in the batch. As a result, they are evaluated only once for all queries in the batch, and thus LMFAO can compute the aggregate batch with minimal interpretation overhead.

We next explain further optimizations that LMFAO employs for this query batch.

First, LMFAO assigns, for each query in the batch, one relation in the join tree as the root for this query. For queries of the form (15.5), we choose as root the largest relation that contains the group-by variable X_i. For queries of the form (15.4), we slightly modify the heuristic defined in Section 12.3, and only consider the functions that define the threshold conditions for X_i, because all other functions are shared across all queries in the batch. As a result, LMFAO again chooses as root the largest relation that contains variable X_i.

This root assignment allows for extensive sharing of computation for all queries in the batch. For instance, any two queries in the batch, which are assigned to the same root R, require the computation of the exact same aggregate over all relations except R. LMFAO can exploit this common structure in the queries by decomposing the queries into views and then sharing and merging views.

This sharing is shown by the example of computing regression trees over the Favorita dataset. We consider 20 threshold conditions for each continuous variable. For each node in the regression tree, we compute a batch of 270 aggregates. LMFAO is able to merge all

of these aggregates into 26 views, which are then computed in 11 view groups. In addition, many of these view groups can be computed in parallel. We show in Chapter 19 that LMFAO can compute the query batch in a few seconds, whereas MonetDB and a commercial data management system take two orders of magnitude longer.

15.3 K-Means

To cluster the dataset defined by the feature extraction query, LMFAO uses the Rk-means algorithm from Section 9.2.2. In total, the algorithm requires the computation of $n + 2$ queries, where $n + 1$ queries are computed in Step 1, and one query is computed in Step 2 of the algorithm.

For Step 1, LMFAO computes, for each variable X_i, a group-by count query that has X_i as a group-by variable. These queries define the subspaces that are then clustered optimally in Step 2. We also need to compute the count over all tuples in the query result.

For Step 3, LMFAO then computes the weighted corset G with another group-by count query, where the group-by variables are given by the cluster assignments from Step 2.

For each query in Step 1, LMFAO chooses as root the largest relation in the join tree that contains the group-by variable of the query. Note that the queries are similar to the queries of the form (15.5), where the aggregate is a simple count instead of a product of functions. As a result, LMFAO can share the majority of the computation across all aggregates in the batch, which makes the computation of the batch very efficient.

In order to avoid that LMFAO generates and compiles separate code for Step 1 and 3, we adjust the problem as follows. For each group-by query in Step 1 with root R, LMFAO extends relation R with an additional variable C_i that defines the cluster assignment for the corresponding subspace. Initially, all variables that define cluster assignments have value zero for all tuples in the relation.

LMFAO then generates the code for Step 1, where the count query is modified into a group-by aggregate query that has the cluster assignment variables as group-by variables. Since all cluster assignments are set to the same value, the result of this query has a single tuple, and the aggregate is equivalent to the count of the number of tuples in the result of the feature extraction query.

Step 2 then clusters each subspace using the optimal clustering algorithms outlined in Section 9.2.2, and LMFAO updates the cluster assignment variables in each relation.

For step 3, LMFAO then recomputes the views for the group-by count query over the cluster assignment variables from Step 1. The result of this query now represents the grid coreset, which is the input to Step 4 of the algorithm.

For the Favorita dataset, LMFAO computes 20 aggregates, and 25 views, which are grouped into 11 view groups.

Chapter 16

Related Work

LMFAO builds on a vast literature of database research. We cited highly relevant work in previous sections. We next mention further connections to work on sharing computation and code compilation. We also overview common low-level optimizations that are commonly employed in machine learning systems. LMFAO computes a batch of group-by aggregates over the same joins without materializing these joins, in the spirit of ad-hoc mining [32], eager aggregation [137], and factorized databases [23].

Sharing Computation

Prior techniques for data cubes use a lattice of sub-queries to capture sharing across the group-by aggregates defining data cubes [65, 91]. Which cells to materialize in a data cube is decided based on space or user-specified constraints [65, 91]. More recent work revisited shared workload optimization for queries with hash joins and shared scans and proposes an algorithm that, given a set of statements and their relative frequency in the workload, outputs a global plan over shared operators [53]. Data Canopy is a library of frequently used statistics in the form of aggregates that can speed up repeating requests for the same statistics. It is concerned with how to decompose, represent, and access such statistics in an efficient manner [135].

Multi-Query Optimization

Multi-Query Optimization (MQO) [122] is concerned with identifying common subexpressions across a set of queries with the purpose of avoiding redundant computation. One of the three types of view merging in LMFAO is also concerned with the same goal, though for directional views with group-by aggregates. LMFAO's view merging proved useful in case of very many and similar views, such as for the applications detailed in Section 15. An alternative type of MQO is concerned with caching intermediate query results, such as in the MonetDB system that we used in experiments.

Code Compilation

Recent work uses code compilation to reduce the interpretation overhead of query evaluation [95, 74, 124, 123, 108, 130]. The compilation approach in LMFAO is closest in spirit to DBLAB [124], which advocates for the use of intermediate representations (IR) to enable code optimizations that cannot be achieved by conventional query optimizers or query compilation techniques without IRs.

The various optimization layers of LMFAO can be viewed as optimizations over the following increasingly more granular IRs: (1) the join tree; (2) orders of join attributes; and (3) the multi-output optimization that registers the computation of aggregates at specific attributes in the attribute order.

LMFAO relies on these IRs to identify optimizations that are not available in conventional query processing. For instance, the join tree is used to identify views that can be grouped and evaluated together as one main computational unit in LMFAO (c.f. Section 12.5). A view group works on a large amount of data at once. This departs from standard query processing that pipelines tuples between relational operators in the execution plan for one query.

Low-Level Optimizations in ML packages

Most ML libraries exploit sparsity in the form of zero-values (due to missing values or one-hot encoding), yet are not structure-aware. LMFAO exploits a more powerful form of sparsity that is prevalent in training datasets defined by joins of multiple relations: This is the join factorization that avoids the repeated representation of and computation over arbitrarily-sized data blocks. LMFAO's code optimizations aim specifically at generating succinct and efficient C++ code for the shared computation of many aggregates over the join of a large table and several views represented as ordered vectors or hashmaps. The layout of the generated code is important: how to decompose the aggregates, when to initialize and update them, how to share partial computation across many aggregates with different group-by and UDAFs (Section 13). Lower-level optimizations (Section 14) are generic and adapted to our workload, e.g., how to manage large amounts of aggregates and how to update them in sequence. LMFAO's multi-aggregate optimizations are absent in ML and linear algebra packages. We next highlight some code optimizations used in these packages. BLAS and LAPACK provide cache-efficient block matrix operations. Eigen [62] supports both dense and sparse matrices, fuses operators to avoid intermediate results, and couples loop unrolling with SIMD vectorization. SPOOF [47] translates linear algebra operations into sum-product form and detects opportunities for aggregate pushdown and operator fusion. LGen [127] uses compilation to generate efficient basic linear algebra operators for small dense, symmetric, or triangular matrices by employing loop fusion, loop tiling, and vectorization. TACO [77] can generate compound linear algebra operations on both dense and sparse matrices. LMFAO can also learn decision trees, which cannot be expressed in

linear algebra. XGBoost [37] is a gradient boosting library that uses decision trees as base learners. It represents the training dataset in a compressed sparse columnar (CSC) format, which is partitioned into blocks that are optimized for cache access, in-memory computation, parallelization, and can be stored on disk for out-of-core learning. LMFAO may also benefit from a combination of value-based compression and factorized representation of the training dataset, as well as from an out-of-core learning mechanism.

Part III

Experimental Evaluation

Chapter 17

Experimental Evaluation Summary

In this part, we present the experimental evaluation of LMFAO. The outcome of these experiments is twofold: (1) Current database management systems cannot efficiently compute large batches of aggregates as required by a variety of machine learning workloads; and (2) scalability challenges faced by state-of-the-art machine learning systems can be mitigated by a combination of database system techniques.

We conducted the following three benchmarks on four datasets, which have size and structure that is common in retail and advertising analytics scenarios. Chapter 18 presents the details of the datasets.

(1) In Chapter 19, we benchmark the computation of batches of aggregates in LMFAO, MonetDB, and DBX (a commercial DBMS) for the following workloads: (1) the covar matrix; (2) a single node in a regression tree; (3) the mutual information of all pairwise combinations of discrete attributes; and (4) a data cube. LMFAO consistently outperforms both DBX and MonetDB on all experiments, with a speedup of up to three orders of magnitude. This shows that MonetDB and DBX cannot efficiently compute large batches of aggregates as required by a variety of analytics workloads. We previously presented these experiments in [118].

(2) In Chapter 20, we benchmark the end-to-end performance of learning predictive models over databases in LMFAO, TensorFlow, MADlib, and scikit-learn. In particular, we consider the following models: (1) linear regression, (2) regression trees, and (3) classification trees. LMFAO again consistently outperforms the competitors, whenever they do not fail due to internal design limitations. In many cases, LMFAO learns the end-to-end model even faster than it takes the competitors to construct the input training dataset. We previously presented a subset of these results in [118].

(3) In Chapter 21, we benchmark the Rk-means clustering algorithm from Section 9.2.2 against mlpack, where Step 1 and Step 3 of the algorithm are computed in LMFAO. The experiments show that the coresets of Rk-means are often significantly smaller

than the data matrix. As a result, we can scale easily to large datasets, and can compute the clusters with a much lower memory footprint than mlpack. The end-to-end performance is orders-of-magnitude faster than mlplack, with minor effects on the approximation. We previously presented these experiments in [42].

In previous publications, we also presented benchmarks for the structure-aware learning of factorization machines and polynomial regression models in the AC/DC system [8, 6]. These results are not presented in this thesis due to space limitations.

Chapter 18

Datasets

We consider four datasets: (1) *Retailer* is used by a large US retailer for forecasting user demands and sales; (2) *Favorita* [49] is a public real dataset that is also used for retail forecasting; (3) *Yelp* is based on the public Yelp Dataset Challenge [138] and contains information about review ratings that users give to businesses; (4) *TPC-DS* [93] (scale factor 10, excerpt) is a synthetic dataset designed for decision support applications. The structure and size of these datasets are common in retail and advertising, where data is generated by sales transactions or click streams. Table 18.1 provides the key characteristics for each dataset, and the join trees used for our experiments are presented in Figure 18.2. We next give a detailed description of each dataset and its schema.

Retailer

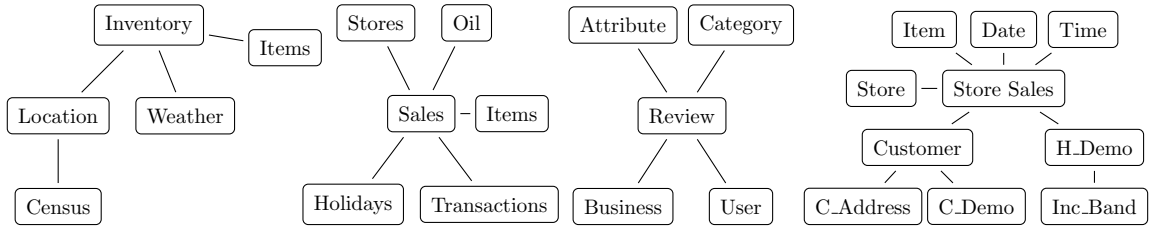
Retailer has five relations: *Inventory* stores the number of inventory units for each date, store, and stock keeping unit (sku); *Location* keeps for each store: its zipcode, the distance to competitors, the store type; *Census* provides 14 attributes that describe the demographics of each zipcode, including the population or median age; *Weather* stores statistics about the weather condition for each date and store, including the temperature or if it rained; *Items* keeps the price, category, subcategory, and category cluster of each sku.

Favorita

Favorita has six relations. Its schema is given in Figure 12.1. *Sales* stores the number of units sold for each store, date, and item, and whether the item was on promotion; *Items* provides information about the skus, such as the item class and price; *Stores* keeps information on stores, like the city they are located it; *Transactions* stores the number of transactions for each date and store; *Oil* provides the oil price for each date; and *Holiday* indicates whether a given date is a holiday.

	Retailer	Favorita	Yelp	TPC-DS
Tuples in Database	87M	125M	8.7M	30M
Size of Database	1.5GB	2.5GB	0.2GB	3.4GB
Tuples in Join Result	86M	127M	360M	28M
Size of Join Result	18GB	7GB	40GB	9GB
Relations	5	6	5	10
Variables	43	18	37	85
Categorical Variables	5	15	11	26
↪ Number of Categories	3.7K	4.5K	1.5M	4.8K

Table 18.1: Key characteristics of the Retailer, Favorita, Yelp, and TPC-DS datasets.



(a) Retailer (b) Favorita (c) Yelp (d) TPC-DS

Figure 18.2: Join Trees for the Retailer, Favorita, Yelp and TPC-DS datasets.

Yelp

Yelp has five relations: *Review* gives the rating and date for each review by users of businesses; *User* keeps information about the users, e.g., how many reviews they made, or when they joined; *Business* provides information about the business, including its location and average rating; *Category* and *Attribute* keep for each business its categories (e.g., Restaurant, Shopping), and respectively attributes, (e.g., open late, has parking). A business can have many attributes and categories.

TPC-DS

TPC-DS [93] is an excerpt of the snowflake query with the Store Sales fact table and scale factor 10. We consider the ten relations and schema shown in Figure 18.2(d). We modified the generated relations by (1) turning strings into integer ids, (2) populating null values, and (3) dropping attributes that are not relevant for our analytics workloads, e.g. street name or categorical attributes with only one category.

Training and Test Datasets

In order to assess the accuracy of the learned models, we separate out test data for each dataset that the model is not trained over. The test data constitutes the sales in the last month in the dataset, for Retailer and Favorita, and the last 15 days for TPC-DS. This simulates the realistic usecase where the ML model predicts future sales.

The training dataset for the machine learning benchmarks is defined by the natural join of the remaining tuples in the input database.

Chapter 19

Aggregate Computation

In this chapter, we report on the performance of LMFAO for the computation of batches of aggregates (Section 19.1). In addition, we provide a breakdown of the performance benefit of various optimization layers in LMFAO for the computation of the covar matrix (Section 19.2).

19.1 Aggregate Computation Benchmark

We benchmark the computation of batches of aggregates in LMFAO, MonetDB, and DBX (a commercial DBMS) for the following workloads and each of the four datasets from Section 18: (1) the non-centered covariance matrix, called the covar matrix; (2) a single node in a regression tree; (3) the mutual information of all pairwise combinations of discrete attributes; and (4) a data cube.

For each workload and dataset, Table 19.1 details how many aggregates, views, and groups are computed. It also gives the size on disk of the aggregates. This is a strong indicator of the running time; except for data cubes, these sizes are much smaller than the result of the underlying join.

19.1.1 Competitors

We benchmarked our system LMFAO, MonetDB 1.1 [70], and DBX (a commercial DBMS). PostgreSQL (PSQL) 11.1 proved consistently slower than DBX and MonetDB, and thus we do not report its performance. We also use LMFAO's predecessor AC/DC [6] as a proxy for computing the covar matrix in an interpreted version of LMFAO without optimizations.

We attempted to benchmark against EmptyHeaded [2], which computes single aggregates over join trees. It, however, requires an extensive preprocessing of the dataset to turn the relations into a specific input format. This preprocessing step introduces significant overhead, which, when applied to our datasets, blows up the size of the data to the extent that it no longer fits into memory. For instance, the Inventory relation in the Retailer

	Retailer					Favorita				
Application	A	I	V	G	Size	A	I	V	G	Size
Covar Matrix	814	654	34	7	0.1	140	46	125	9	0.9
Regression Tree Node	3141	16	19	9	0.1	270	20	26	11	0.1
Mutual Information	56	22	78	8	76	106	35	141	9	10
Data Cube	40	8	12	5	3944	40	7	13	6	5463

	Yelp					TPC-DS				
Application	A	I	V	G	Size	A	I	V	G	Size
Covar Matrix	730	309	99	8	1795	3061	590	286	14	577
Regression Tree Node	1392	16	22	9	39	4299	138	52	17	0.2
Mutual Information	172	64	236	9	1759	301	95	396	15	55
Data Cube	40	7	13	5	1876	40	12	17	10	3794

Table 19.1: Number of application aggregates (A), additional intermediate aggregates synthesised by LMFAO (I), views (V), and groups of views (G) for each dataset and aggregate batches: covar matrix (CM), regression tree node (RT), mutual information (MI), and data cube (DC). The size on disk of the application aggregates is given in MB.

dataset (2GB) is blown up to more than 300GB during preprocessing. Our observation is that EmptyHeaded has difficulty preprocessing relations whose arity is beyond two. We were therefore unable to compare against EmptyHeaded.

19.1.2 Experimental Setup

We run all experiments on a dedicated AWS d2.xlarge instance with Ubuntu 18.04 and four vCPUs. We used the O3 compiler optimization flag and report wall-clock times by running each system once and then reporting the average of four subsequent runs with warm cache. We do not report the times to load the database into memory. All relations are given sorted by their join attributes.

The covar matrix and regression tree node are computed over all attributes in Yelp, all but the join keys in Retailer and TPC-DS, and all but date and item in Favorita. We compute all pairwise mutual information aggregates over nine attributes for Retailer, 15 for Favorita, 11 for Yelp, and 19 for TPC-DS. These attributes include all categorical and some discrete continuous attributes in each dataset. For data cubes, we used three dimensions and five measures for all experiments. We provide DBX and MonetDB with the same list of queries as LMFAO, which may have multiple aggregates per query.

19.1.3 Takeaways

Figure 19.2 presents the performance of each system for the four workloads. The bars for DBX and MonetDB are annotated with the relative speedup of LMFAO over each of them.

LMFAO consistently outperforms both DBX and MonetDB on all experiments, with a

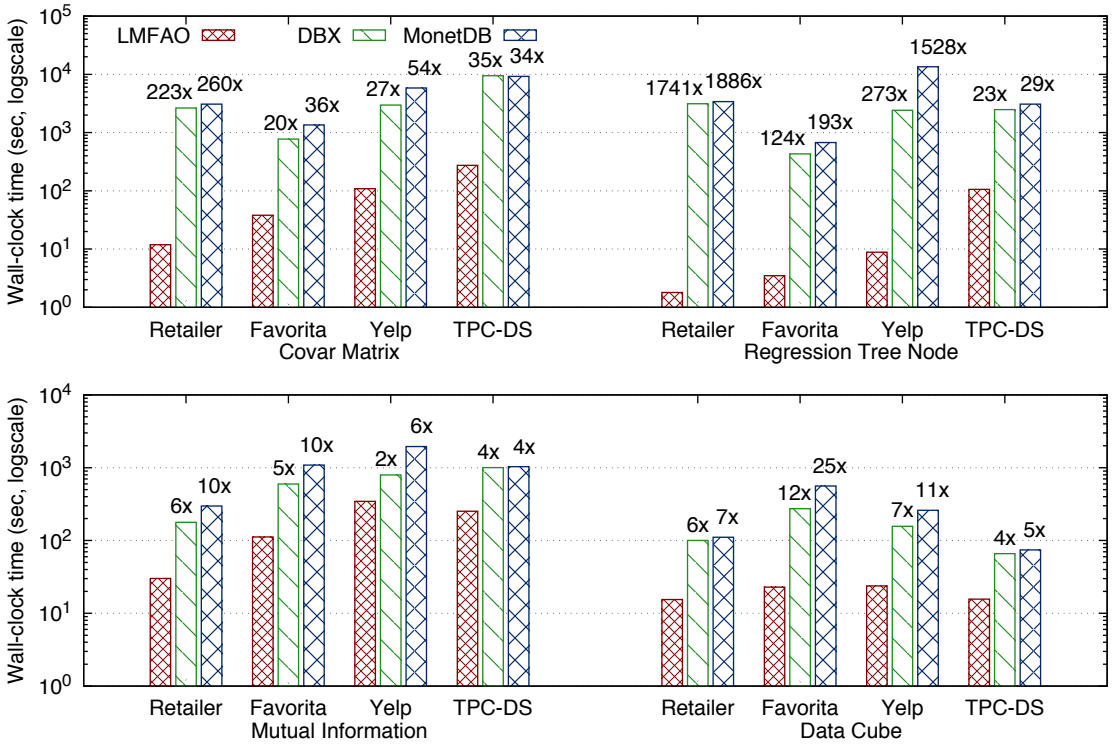


Figure 19.2: Time performance for computing various batches of aggregates using LMFAO, MonetDB, and DBX. The bars are annotated with the relative speedup of LMFAO over MonetDB and DBX.

speedup of up to three orders of magnitude. The reason is as follows. Whereas LMFAO shares computation across aggregates, DBX and MonetDB compute each query in the batch as an individual query and thus fail to share computation across aggregates.

We assess how much each system shares computation across aggregates in a batch by using the time to compute a single count query as proxy. Table 19.3 presents the performance of each system for the count query. For instance, the covar matrix for Retailer has 814 aggregates. Without sharing, the performance would be at least 814× that of the count query, or 6510 seconds. The performance of LMFAO is, however, 55× better! In contrast, the performance of computing the covar matrix in DBX and MonetDB is not faster than computing the count query 814 times. Whereas the performance of MonetDB is roughly equal to the time it takes to compute the count query 814 times, the performance of DBX is 1.4× *slower*. Therefore, we conclude that DBX and MonetDB fail to share computation for this workload.

We next discuss how LMFAO can achieve orders of magnitude speedup in more detail.

LMFAO clusters the query batch into a few groups that are computed together in a single pass over the fact table and at most two passes over the smaller dimension tables. The fact table in Retailer has few variables and thus most aggregates are computed over the dimension tables. In comparison, Favorita requires relatively few aggregates, and the

Count	Retailer		Favorita		Yelp		TPC-DS	
LMFAO	0.80	1.00×	0.97	1.00×	0.68	1.00×	5.01	1.00×
DBX	2.38	2.98×	4.04	4.15×	2.53	3.72×	2.84	0.57×
MonetDB	3.75	4.70×	8.11	8.32×	4.37	6.44×	2.84	0.57×

Table 19.3: Time performance (seconds) for computing a single count aggregate using LMFAO, MonetDB, and DBX and relative speedup of LMFAO over MonetDB and DBX.

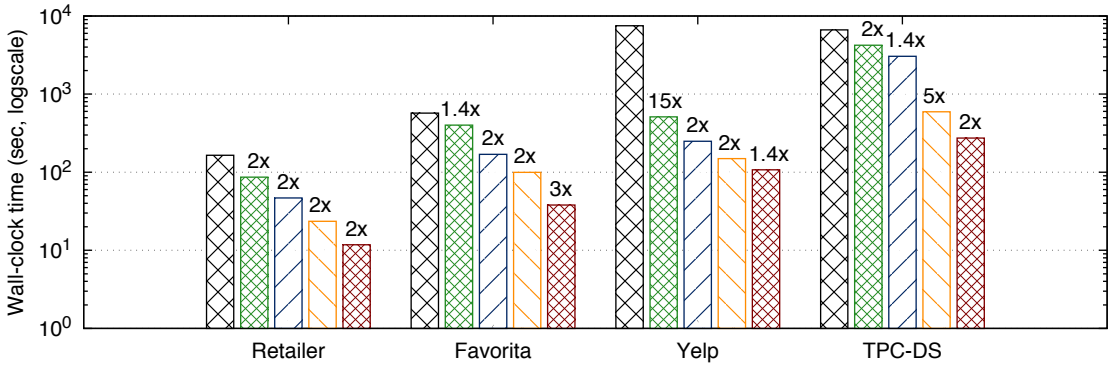


Figure 19.4: Performance impact of optimizations in LMFAO for computing the covar matrix. From left to right: no optimization (time for AC/DC proxy shown); compilation; plus multi-output; plus multi-root; and plus parallelization with four threads. Bars are annotated with relative speedup over the previous bar.

large fact table in TPC-DS has many variables, which means many aggregates are computed over it. This explains the relatively lower performance improvement for Favorita and TPC-DS. For Yelp, the main reason for the performance gap is that LMFAO's decomposition of aggregates into views avoids the materalization of the many-to-many joins.

The performance gap is particularly large for regression tree nodes. There are two reasons for this gap: (1) in contrast to the other workloads, regression tree nodes do not require queries with group-by variables from different relations, and (2) all queries that are rooted at the same relation share the computation of all views that are computed towards that root. This implies that LMFAO can efficiently share the majority of computation across all aggregates in the batch. In addition, most of the computation is pushed to the small dimension tables, where the aggregates are grouped and computed in one pass over the join of the relation and incoming views that are common for all queries in the group.

19.2 LMFAO Optimization Breakdown

Figure 19.4 shows the performance benefit of LMFAO optimizations for the computation of the covar matrix over each of the four datasets. The baseline is LMFAO's predecessor AC/DC (leftmost bar), a proxy for LMFAO without optimizations. The following bars show the performance of LMFAO with the following optimizations enabled one-by-one: (1)

compilation, (2) multiple roots, (3) multiple-output, (4) parallelization with four threads.

LMFAO with compilation but without the other optimizations achieves a speedup of $1.4 - 15\times$ over AC/DC. Multi-output and multiple roots together further improve the performance by $4 - 7\times$ over LMFAO with compilation. Parallelization with four cores further improves the performance by $1.4 - 3\times$.

Compilation Overhead

The compilation overhead of LMFAO depends on the workload. Using g++6.4.0 and eight cores, it ranges from 2 seconds for data cubes over Favorita to 50 seconds for the mutual information batch over TPC-DS. This overhead is not reported in Figure 19.2 (we report the average of four subsequent runs). It can be reduced using LLVM code generation and compilation [95].

Chapter 20

Learning Predictive Models

In this chapter, we report on the end-to-end performance of LMFAO for the learning of linear regression models (Section 20.1), as well as regression and classification trees (Section 20.2).

20.1 Learning Regression Models

We report on the end-to-end performance of LMFAO for learning ridge linear regression models. The models are computed over Retailer and Favorita, and used to predict the number of inventory units and respectively number of units sold.

20.1.1 Competitors

We benchmark LMFAO against three machine learning packages commonly used in data science: TensorFlow 1.14 (compiled with AVX optimization enabled) [1], MADlib 1.8 [67], and scikit-learn 0.21 [106].

LMFAO first computes the covar matrix and then optimizes the parameters over it using batch gradient descent with Armijo backtracking line search and Barzilai-Borwein step size [24], as described in Section 6.2.

MADlib computes the closed form solution of the model with ordinary least squares. (OLS is the fastest approach supported by MADlib for this problem.) MADlib is integrated with PostgresSQL 10 (PSQL). As a result, we can compute the model over the non-materialized view of the training dataset. MADlib, however, requires that the input data is one-hot encoded.

We tuned PSQL for in-memory processing by setting its working memory to 28GB, shared buffers to 128MB, and turning off the parameters fsync, synchronous commit, full page writes, and bgwriter LRU maxpages.

TensorFlow and scikit-learn require that the training dataset is materialized. Since TensorFlow and scikit-learn do not support query processing tasks, we materialize the feature extraction query that defines the training dataset in Pandas DataFrames [86].

Scikit-learn requires that the training dataset is kept in memory, and imposes the following two restrictions on the input: (1) The training dataset cannot have any variables that are not features, and the label and features need to be provided separately. This means that we have to separate the label from the Pandas DataFrame that stores the join result, and also drop any join variables that are not features in the model. This is a time-consuming intermediate step, and Pandas requires a temporary copy of the training dataset for this operation. (2) Scikit-learn also requires the input to be one-hot encoded, for which we use the OneHotEncoder provided by scikit-learn. In addition to the two transformations, Pandas and scikit-learn require that all values have the same type, so they choose the most general type: Floats. The combination of the above shortcomings has significant overhead, which leads to memory errors for scikit-learn.

TensorFlow learns the linear regression model with the predefined LinearRegressor estimator, which ftrl-optimization [87], a variant of stochastic gradient descent. We report the time to perform one epoch with a batch size of 100,000 instances. This batch size gave the best performance/accuracy trade-off.

TensorFlow does not require the explicit one-hot encoding of the dataset, and encodes categorical variables on the fly during learning. In our experiments, TensorFlow nevertheless runs out of memory for learning over the full Retailer dataset. In this case, we wrote the data to a file and used the CSV parser to iteratively load mini-batches of data during learning. We do not include the time it takes to write the dataset to the file. The CSV parser allows TensorFlow to compute models over large datasets, but it comes with large overhead, as it needs to repeatedly load, parse and cast the batches of tuples. This leads to poor performance for learning models over large datasets.

20.1.2 Experimental Setup

All experiments are performed on an Intel(R) Core(TM) i7-4770 3.40GHz/64bit/32GB with Linux 3.13.0/g++6.4.0 and eight cores. For all experiments, we use a timeout of 24 hours.

We consider three variants of both our Retailer and Favorita datasets:

(1) Variant v1 represents a partition of each dataset, whose size is 25% of the overall dataset. This variant was designed to work within the memory limitations of scikit-learn. We learn linear regression models over all attributes but join keys for Retailer, and all but date and item for Favorita.

(2) Variant v2 learns the model with the same features as v1, but over the complete dataset. This variant is the dataset that is also used in the aggregate computation benchmark in Chapter 19.

(3) Variant v3 extends the models from v2 with the additional variable item, which is a high dimensional categorical variable.

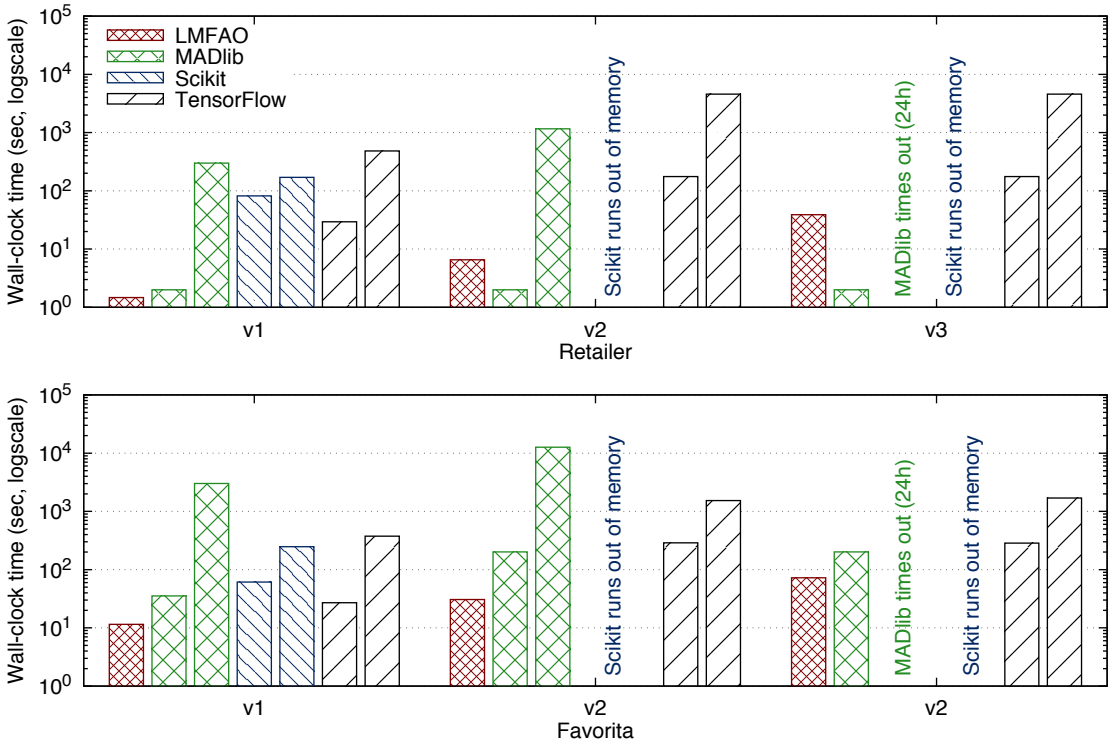


Figure 20.1: Time performance of LMFAO, MADlib, scikit-learn, and TensorFlow for learning linear regression models over Retailer and Favorita.

We evaluate the accuracy of the model by computing the root-mean-square-error (RMSE) over the test dataset, and by ensuring that it is the same for both LMFAO's model and MADlib's closed form solution.

We also attempted to benchmark the learning of degree-2 polynomial regression models, but all competitors either do not support interactions between two variables, run out of memory, or time out after 24h. For this reason, we do not show the results. The full details on the experiments on polynomial regression are presented in [8].

20.1.3 Takeaways

Figure 20.1 shows the performance of the end-to-end learning of linear regression models in LMFAO, MADlib, scikit-learn, and TensorFlow. For MADlib, scikit-learn and TensorFlow, we show two bars. The left bar represents the time it takes to prepare the input dataset for each system. For scikit-learn and TensorFlow, this includes the time it takes to materialize the join result in Pandas. For scikit-learn, it also includes the one-hot encoding of categorical variables. MADlib only requires the one-hot encoding of the input database. The right bar gives the time it takes to learn the model. The end-to-end performance of learning the model is given by the addition of the two bars. LMFAO learns the model directly over the input database, and does not require any preprocessing.

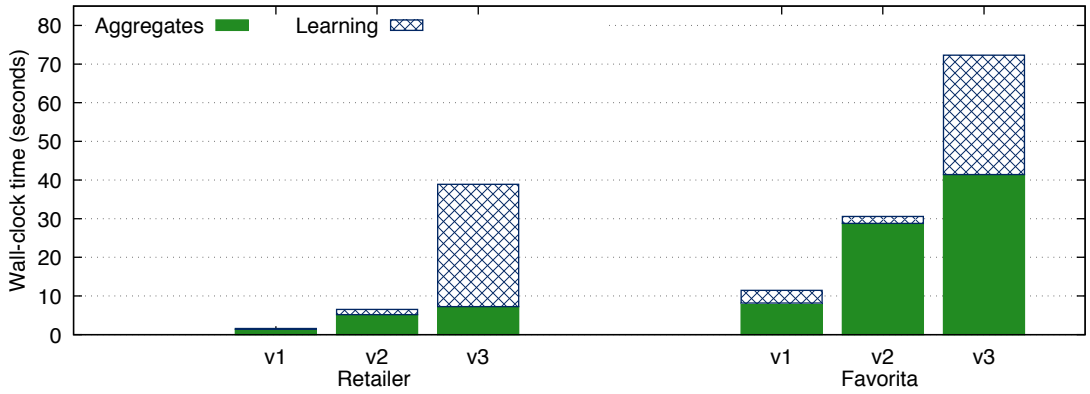


Figure 20.2: Breakdown of the performance of LMFAO for the end-to-end learning of linear regression models for the three variants of Retailer and Favorita. We show the performance of (1) computing the aggregate batch that defines the covar matrix, and (2) learning the model over the aggregates.

The RMSE of the linear regression models for v1 and v2 of Retailer and Favorita in LMFAO is within 1% of that for the closed form solutions computed in MADlib. By extending the models with the categorical variable item (version v3 for both datasets), the RMSE decreases by 21% for Retailer and by 6% for Favorita. MADlib fails to learn these more accurate models, because it times out after 24 hours.

For TensorFlow, the models are trained with a single epoch. The resulting models have a consistently higher RMSE than the corresponding model computed in LMFAO. In particular, for Retailer v3, the RMSE of the TensorFlow model is 31% higher. TensorFlow would require additional epochs to achieve the same accuracy as LMFAO.

We make the following remarkable observations:

(1) State-of-the-art machine learning packages do not scale well for large datasets. Scikit-learn runs out of memory for all our experiments on the large variants of Retailer and Favorita. Madlib times out after 24 hours for v3 of Retailer and Favorita.

(2) LMFAO is able to compute all models orders-of-magnitude faster than the competitors. In particular, LMFAO learns the model over the input database even faster than the preprocessing required by TensorFlow and scikit-learn. This is because LMFAO avoids the materialization of the large training dataset and works directly on the input database: For Retailer, the former is 10× larger than the latter (Table 18.1). Furthermore, for linear regression, the convergence step takes as input the result of the aggregate batch, which is again at least an order of magnitude smaller than the input database.

(3) In contrast to MADlib and scikit-learn, LMFAO is able to efficiently accommodate high-dimensional categorical variables. This is becuase LMFAO does not require the

explicit one-hot of the variable, and naturally accommodates for a sparse encoding of categorical variables in the covar matrix. TensorFlow's feature mapping interface also allows it to accommodate for high-dimension categorical variables, but this comes with significant overhead. As a result, TensorFlow computes a single epoch orders-of-magnitude slower than LMFAO.

Figure 20.2 shows the breakdown of the LMFAO performance into the computation of the aggregate batch and the learning task. For the aggregate computation, the performance decreases by a factor of four as we go from v1 to v2, which is linear in the increase of the dataset size. The addition of the categorical variable item in v3 does not significantly affect the performance of the aggregate computation. This is because LMFAO can efficiently share computation of the additional aggregates that are required.

LMFAO learns the model parameters in a few seconds for variants v1 and v2 for both datasets. The version v3 has significantly more parameters and LMFAO requires around 30 seconds to converge. This result has remarkable implications for model selection. LMFAO can compute the aggregate batch for the covar matrix only once, and then learn any model over a subset of the features in v3 on the covar matrix in 3-30 seconds. In contrast, our competitors would require hours for each model, whenever they do not fail due to internal design limitations.

20.2 Learning Decision Trees

We report on the end-to-end performance of LMFAO for learning regression trees over the three variants of Retailer and Favorita, and for learning classification trees over TPC-DS. For the latter, we predict whether a customer is a preferred customer, as proposed in the Relational Dataset Repository [90].

20.2.1 Experimental Setup

This benchmark is performed on the same machine, and with the same competitors as the linear regression benchmark in the previous section.

All systems learn the decision trees with the CART algorithm [28]. As cost function, we use the variance for regression trees and the Gini index for classification trees. For TPC-DS, we learn classification trees over all attributes but join keys.

We set the maximum depth of the decision trees to four (i.e. at most 31 nodes), and the minimum number of instances required to split a node to 1000. Continuous attributes are bucketized into 20 buckets. We verify that LMFAO learns decision trees that have the same accuracy as the decision trees learned in MADlib. The runtime for LMFAO includes the time it takes to compute the buckets.

To learn the trees in TensorFlow, we use the built-in BoostedTrees estimator with a batch size of 100K. For continuous attributes, TensorFlow requires the buckets as input,

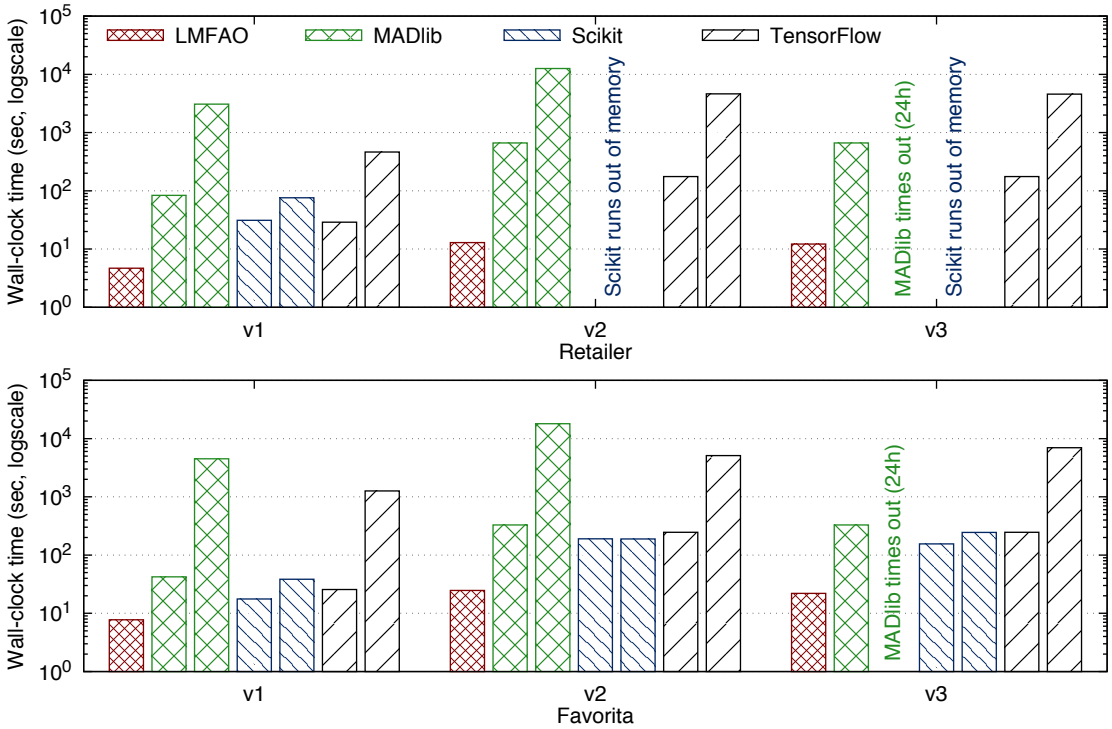


Figure 20.3: Time performance of LMFAO, MADlib, scikit-learn, and TensorFlow for learning regression trees over Retailer and Favorita.

and we provide it with the same buckets as LMFAO.

Since the CART algorithm requires several iterations over the training dataset, MADlib benefits if the traning dataset is explicitly materialized, and not provided as non-materialized view. Therefore, the preprocessing time for MADlib now includes the time it takes PSQL to materialize the training dataset.

Contrary to the linear regression models, scikit-learn does not require the categorical variables to be one-hot encoded to learn decision trees.

20.2.2 Takeaways

Figure 20.3 shows the results of the end-to-end benchmarks for learning regression trees over the three variants of Retailer and Favorita. Figure 20.4 shows the results for learning classification trees over TPC-DS.

As for the linear regression benchmark, we show the performance for MADlib, scikit-learn, and TensorFlow with two bars. The left bar represents the time it takes to join the relations and any preprocessing on the input to the respective competitor. The right bar represents the learning time.

Scikit-learn runs out of memory, when it computes the decision trees over the v1 and v2 of Retailer and over TPC-DS. MADlib times out after 24h for variant v3 of both Retailer and Favorita.

LMFAO learns decision trees with the same accuracy orders-of-magnitude faster than its competitors. For Retailer and Favorita, it even learns the regression tree end-to-end faster than it takes the competitors to materialize the training dataset.

The reason for this performance is twofold. First, LMFAO does not require the materialization of the join result and runs directly on the input database, which is up to an order-of-magnitude smaller (cf. Table 18.1). Second, LMFAO efficiently computes the aggregate batches required to compute each node in a decision tree. This is due to several optimizations in LMFAO, as outlined in the discussion in Section 15.2 and the aggregate computation benchmark from Section 19.1.

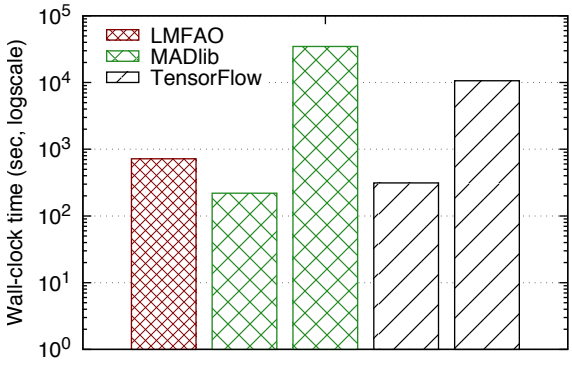


Figure 20.4: Time performance of LMFAO, MADlib, and TensorFlow for learning classification trees over TPC-DS; scikit-learn runs out of memory.

Chapter 21

K-Means Clustering

We evaluate the performance of the Rk-means clustering algorithm from Section 9.2.2, where the steps 1 and 3 of the algorithm are computed in LMFAO (c.f. Section 15.3). We conduct the following two sets of experiments:

1. In Section 21.1, we benchmark the performance and approximation of Rk-means against mlpack [41].

2. In Section 21.2, we break down and analyze the performance of each step in Rk-means, and we evaluate the performance and approximation of Rk-means for setting $\kappa < k$; i.e., different number of clusters for Steps 2 and 4.

21.1 Comparison with mlpack

We consider the Retailer, Favorita, and Yelp datasets. The experiments show that the coresets of Rk-means are often significantly smaller than the data matrix. As a result, Rk-means can scale easily to large datasets, and can compute the clusters with a much lower memory footprint than mlpack. When $\kappa = k$, Rk-means is orders-of-magnitude faster than the end-to-end computation for mlpack—up to 115×. Typically, the approximation level is very minor. In addition, setting $\kappa < k$ can lead to further performance speedups with only a moderate increase in approximation, giving over 200× speedup in some cases.

21.1.1 Experimental Setup.

All experiments were performed on an AWS `x1e.8xlarge` system, which has 1 TiB of RAM and 32 vCPUs.

We consider the Retailer, Favorita, and Yelp datasets from Chapter 18. For Retailer and Favorita, the data matrix to be clustered

	Retailer	Favorita	Yelp
$\kappa = 5$	1.4M	14.9K	2.7M
$\kappa = 10$	9.6M	85.9K	11.7M
$\kappa = 20$	38.3M	632.5K	11.9M
$\kappa = 50$	73.8M	7.9M	12.5M

Figure 21.1: Number of rows in coreset G for different κ.

is defined by the natural join over all relations. The Favorita dataset is slightly modified: The original dataset gave the `units_sold` variable with a precision of three decimals places, which means that the active domain for this variable has many distinct values. This has a significant impact on the Step 2 of the Rk-means algorithm. We decreased the precision for this variable to two decimal places, which lowers the number of distinct values by a factor of four. This modification has no effect on the final clusters or their accuracy.

For Yelp, we consider a smaller variant of the dataset, where the *Attribute* relation is aggregated. This relation thus returns for each business the number of attributes that have been assigned to this business, and does not enumerate over all distinct attributes. The join result of this variant of Yelp has 22M tuples and takes up 2.4GB on disk.

Figure 21.1 presents the size of the coreset G for each dataset and different κ-values. $|G|$ is highly data dependent. For *Favorita*, G is orders-of-magnitude smaller than the data matrix. In contrast, for *Retailer*, when $k = 20$ and $k = 50$, $|G|$ approaches the size of the data matrix D. For *Yelp*, $|G|$ is roughly half the size of D for $k \geq 10$.

We benchmark Rk-means on LMFAO against mlpack [41] (v. 3.1.0), which is a fast C++ machine learning library. Although mlpack supports many machine learning models today, one primary motivation for mlpack was the efficient support for clustering algorithms. This makes mlpack an excellent benchmark for LMFAO.

Mlpack requires that the data matrix is one-hot encoded, and thus requires a lot of memory. To try and reduce RAM usage, we also benchmarked against mlpack with sparse matrices from the Armadillo library [117]. This did reduce RAM usage, but the overhead of working with sparse data structures meant an overall slowdown. Therefore, we only report the performance for mlpack on dense data representations.

We use the seminal k-means++ algorithm [18] to initialize the k-means cluster for both mlpack and Rk-means.

We use PostgreSQL 11.1 (PSQL) to materialize the data matrix D that forms the input to mlpack. We run Rk-means and mlpack + PSQL five times and report the average approximation and runtime. Our runtime results omit data loading/saving times. In the end-to-end pipeline, PSQL must export D to disk, and mlpack must then read it from disk, which adds significant overhead. LMFAO does not require this, and thus the runtime numbers are skewed in mlpack's favor.

21.1.2 Takeaways

The left columns of Table 21.2 compares the runtime and approximation of Rk-means against mlpack on the three datasets for different k values with $\kappa = k$. The approximation is given relative to the objective value obtained by mlpack. Speedup is given by comparing the end-to-end performance of Rk-means and mlpack (ignoring disk I/O time), which for mlpack includes the time needed by `psql` to materialize D.

Rk-means outperforms even just the clustering step from mlpack, and when end-to-end

Retailer	k = 5	k = 10	k = 20	k = 50	k=20,κ=10	k=50,κ=20
Compute D (psql)	175.47	175.47	175.47	175.47	175.47	175.47
Clustering (mlpack)	65.41	158.81	385.67	1,453.88	385.67	1,453.88
Rk-means	15.66	54.59	230.17	650.20	63.51	344.31
Relative Speedup	15.38×	6.12×	2.44×	2.51×	8.84×	4.73×
Relative Approx.	0.20	0.08	0.03	0.00	0.03	0.02
Favorita	k = 5	k = 10	k = 20	k = 50	k=20,κ=10	k=50,κ=20
Compute D (psql)	156.86	156.86	156.86	156.86	156.86	156.86
Clustering (mlpack)	1,002.54	6,449.32	11,794.49	43,011.67	11,794.49	43,011.67
Rk-means	27.95	57.72	118.36	334.65	57.65	120.77
Relative Speedup	41.49×	114.59×	100.98×	129.00×	207.30×	357.44×
Relative Approx.	2.99	0.35	0.12	0.15	1.93	1.92
Yelp	k = 5	k = 10	k = 20	k = 50	k=20,κ=10	k=50,κ=20
Compute D (psql)	33.83	33.83	33.83	33.83	33.83	33.83
Clustering (mlpack)	210.59	640.43	2,107.83	11,474.24	2,107.83	11,474.24
Rk-means	43.37	107.71	195.22	405.11	114.34	241.34
Relative Speedup	5.64×	6.26×	10.97×	28.41×	18.73×	47.68×
Relative Approx.	0.37	0.26	0.13	0.05	0.27	0.20

Table 21.2: End-to-end runtime and approximation comparison of Rk-means over LMFAO and mlpack on each dataset. The first four columns use different $\kappa = k$ values; the last two show results for setting $\kappa < k$.

computation is considered, Rk-means gives up to 129× speedup. For large datasets and k, like *Favorita* with $k = 50$, the time to cluster is extremely large (nearly 12 hours!) while Rk-means is able to obtain a result with minimal approximation in just over 5 minutes. In addition, Rk-means has a much smaller memory footprint than mlpack: for instance, on the *Favorita* dataset with $k = 20$, mlpack uses over 900GiB of RAM to cluster the dataset, whereas Rk-means only requires 18Gib. In our simulations, the approximation level is moderate, and consistently well below the 9-approximation bound of Rk-means.

21.2 Breakdown of Rk-means

Figure 21.3 shows the time it takes Rk-means to cluster the three datasets for different values of k with $\kappa = k$. The total time is broken down into the four steps of the algorithm from Section 9.2.2. We provide the time it takes PSQL to compute D as reference (white bar). In many cases, Rk-means can cluster *Retailer* and *Favorita* faster than it takes PSQL to even compute the data matrix! The relative performance of the four steps is data dependent. For *Retailer*, most of the time is spent on constructing G in Step 3, which is relatively large. For *Favorita*, however, Step 2 takes the longest, since it contains one continuous variable with many distinct values, and the DP algorithm for clustering runs in time quadratic in the number of distinct values. The runtime for *Favorita* could be improved by clustering this dimension with a different k-means algorithm instead, but this may increase the approximation.

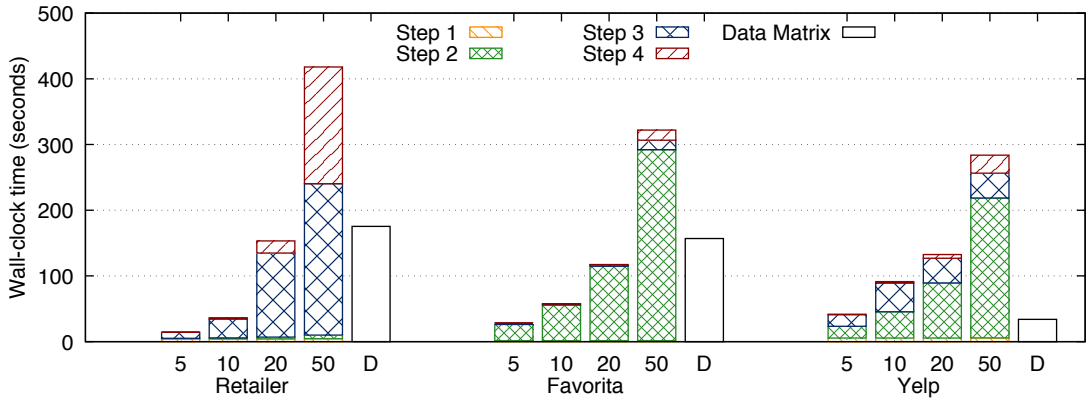


Figure 21.3: Breakdown of the compute time of Rk-means for each step of the algorithm with $\kappa = k$. The time to compute data matrix D is provided as reference.

21.2.1 Setting $\kappa < k$ for Step 2

We next evaluate the effect of setting κ to a smaller value than the number of clusters k. This exploits the speed/approximation tradeoff: smaller κ helps reduce the size of $\boldsymbol{G}$, at the cost of more approximation. Table 21.2 presents for each dataset the results for setting $k = 20, \kappa = 10$ and $k = 50, \kappa = 20$, and compares them to the relative performance and approximation over computing k clusters in mlpack.

By setting $\kappa < k$, Rk-means can compute the k clusters up to 357× faster than mlpack and 3.6× faster than Rk-means with $\kappa = k$, while the relative approximation remains moderate. Our results are data dependent—but as queries and databases scale, our speedups will be even more significant.

Chapter 22

Conclusion and Future Work

This thesis puts forward a new regime for the computation of machine learning models over multi-relational databases. We avoid the costly repeated loop that the mainstream approaches require: select features from data residing in relational databases using feature extraction queries; export the training dataset defined by such queries; convert this dataset into the format of the machine learning tool; and then train the desired model using this tool. Instead, we present a highly integrated solution that merges the computation of the feature extraction query and the learning task.

The main observation is that the data-intensive computation of the learning task for a variety of models can be expressed as large batches of aggregate queries. As a consequence, we can inspect the problem through the lens of database theory. In particular, we can employ recent developments in database query evaluation, which exploit structure in the query and data to optimize the computation of the aggregate queries. Based on these known techniques, we present novel results for the computation of aggregate queries with additive inequalities, which have lower asymptotic runtime than existing techniques and have many applications in machine learning.

As a consequence of the theoretical investigation of the problem, we merge the computation of the query and learning tasks by pushing the data-intensive computation of the learning task past the join in the feature extraction query. We prove that, for a large class of feature extraction queries and machine learning models, we can achieve asymptotic runtime improvements over the mainstream approaches that require the query to be materialized. This class of models includes linear regression, polynomial regression, support vector machines, factorization machines, principal component analysis, regression and classification trees, and k-means clustering.

We further present LMFAO (Layered Multiple Functional Aggregate Optimization), an in-memory execution engine for batches of aggregates over relational data. LMFAO consists of several layers of logical and code optimizations that systematically exploit factorization, sharing of computation, parallelism, and code specialization.

An experimental evaluation of LMFAO has shown that the optimizations are very effec-

tive for computing large query batches, as well as the end-to-end learning over databases. On the one hand, we outperform MonetDB and a commerical DBMS by orders of magnitude for the computation of aggregate batches. On the other hand, LMFAO significantly outperforms ML libraries including scikit-learn, TensorFlow, MADlib, and mlpack. These results confirm our two main observations: (1) Current database management systems cannot efficiently compute large batches of aggregates as required by a variety of machine learning workloads; and (2) scalability challenges faced by state-of-the-art machine learning systems can be mitigated by a combination of database systems techniques.

We believe that the results presented in this thesis only scratch the surface of the potential that lies within exploiting the relational structure in the learning of models over databases. We next present a short overview of open problems.

One research direction is to extend the class of machine learning models that can be trained efficiently by exploiting the structure of the underlying relational database. In particular, we plan to investigate the following model classes in future work: deep neural networks, boosted decision trees, kernel methods, and latent variable models. We have promising preliminary results for the learning of sum-product networks and boosted decision trees, and many more will certainly follow.

Another interesting research direction concerns the impact of functional dependencies (FD) on the learning of the model. Preliminary work has shown that linear FDs among model features can be exploited to reduce the dimensionality of the learning problem [6, 80]. It would be interesting to extend these results to general linear dependencies, and also to a larger class of models. In addition, the statistical implications of FD-based dimensionality reduction of the feature space have not been studied extensively.

On the systems side, there are several extensions to LMFAO that present interesting and fruitful research directions. In particular, we would like to extend LMFAO with (1) an application layer that can automatically translate any machine learning problem into a batch of aggregate queries, (2) efficient support of additive inequalities, and (3) the ability to exploit functional dependencies to optimize the model as well as the computation of the aggregate batches.

For the first extension, we are in the process of developing a front end to LMFAO, which takes as input a program that defines an optimization algorithm to learn a model. The engine then optimizes this program using a combination of novel and known programming languages techniques that identify chunks of code that can be cast into batches of aggregate queries. These queries are then optimized and computed by LMFAO. Therefore, we combine the database-centric optimizations in LMFAO with optimizations from the programming language community. We present preliminary results on this extensions in [125].

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *OSDI*, pages 265–283, 2016.

[2] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. In *SIGMOD*, pages 431–446, 2016.

[3] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[4] Serge Abiteboul, Marcelo Arenas, Pablo Barceló, Meghyn Bienvenu, Diego Calvanese, Claire David, Richard Hull, Eyke Hüllermeier, Benny Kimelfeld, Leonid Libkin, Wim Martens, Tova Milo, Filip Murlak, Frank Neven, Magdalena Ortiz, Thomas Schwentick, Julia Stoyanovich, Jianwen Su, Dan Suciu, Victor Vianu, and Ke Yi. Research directions for principles of data management (Dagstuhl Perspectives Workshop 16151). *CoRR*, abs/1701.09007, 2017.

[5] Mahmoud Abo Khamis, Ryan R. Curtin, Benjamin Moseley, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. On functional aggregate queries with additive inequalities. In *PODS*, pages 414–431, 2019.

[6] Mahmoud Abo Khamis, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. AC/DC: In-database learning thunderstruck. In *DEEM*, pages 8:1–8:10, 2018.

[7] Mahmoud Abo Khamis, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. In-database learning with sparse tensors. In *PODS*, pages 325–340, 2018.

[8] Mahmoud Abo Khamis, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. Learning models over relational data using sparse tensors and functional dependencies. To appear in ACM TODS 2020, *CoRR*, abs/1703.04780, 2019.

[9] Mahmoud Abo Khamis, Hung Q. Ngo, Dan Olteanu, and Dan Suciu. Boolean tensor decomposition for conjunctive queries with negation. In *ICDT*, 2019.

[10] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. FAQ: questions asked frequently. In *PODS*, pages 13–28, 2016.

[11] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. Juggling functions inside a database. *SIGMOD Rec.*, 46(1):6–13, 2017.

[12] Rakesh Agrawal, Heikki Mannila, Ramakrishnan Srikant, Hannu Toivonen, and A. Inkeri Verkamo. Fast discovery of association rules. In Usama M. Fayyad, Gregory Piatetsky-Shapiro, Padhraic Smyth, and Ramasamy Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. American Association for Artificial Intelligence, 1996.

[13] Sara Ahmadian, Ashkan Norouzi-Fard, Ola Svensson, and Justin Ward. Better guarantees for k-means and euclidean k-median by primal-dual algorithms. In *FOCS*, pages 61–72, 2017.

[14] Srinivas M Aji and Robert J McEliece. The generalized distributive law. *IEEE transactions on Information Theory*, 46(2):325–343, 2000.

[15] Hirotugu Akaike. A new look at the statistical model identification. *IEEE Transactions on Automatic Control*, AC-19(6):716–723, December 1974.

[16] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design and implementation of the LogicBlox system. In *SIGMOD*, pages 1371–1382, 2015.

[17] Larry Armijo. Minimization of functions having lipschitz continuous first partial derivatives. *Pacific Journal of mathematics*, 16(1):1–3, 1966.

[18] D. Arthur and S. Vassilvitskii. k-means++: The advantages of careful seeding. In *SODA*, page 1027–1035, 2007.

[19] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. In *FOCS*, pages 739–748, 2008.

[20] Olivier Bachem, Mario Lucic, and Andreas Krause. Practical coreset constructions for machine learning. *arXiv preprint arXiv:1703.06476*, 2017.

[21] Olivier Bachem, Mario Lucic, and Andreas Krause. Scalable k -means clustering via lightweight coresets. In *SIGKDD*, pages 1119–1127, 2018.

[22] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *CSL*, pages 208–222, 2007.

[23] Nurzhan Bakibayev, Tomás Kociský, Dan Olteanu, and Jakub Závodný. Aggregation and ordering in factorised databases. *PVLDB*, 6(14):1990–2001, 2013.

[24] Jonathan Barzilai and Jonathan M. Borwein. Two-point step size gradient methods. *IMA Journal of Numerical Analysis*, 8(1):141–148, 1988.

[25] Matthias Boehm, Shirish Tatikonda, Berthold Reinwald, Prithviraj Sen, Yuanyuan Tian, Douglas Burdick, and Shivakumar Vaithyanathan. Hybrid parallelization strategies for large-scale machine learning in SystemML. *PVLDB*, 7(7):553–564, 2014.

[26] Léon Bottou. Stochastic gradient descent tricks. In *Neural Networks: Tricks of the Trade (2nd ed)*, pages 421–436, 2012.

[27] Jean-Francois Boulicaut and Cyrille Masson. Data mining query languages. In Oded Maimon and Lior Rokach, editors, *Data Mining and Knowledge Discovery Handbook*, pages 715–726. Springer, Boston, MA, 2005.

[28] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA, 1984.

[29] Robert Brijder, Floris Geerts, Jan Van den Bussche, and Timmy Weerwag. On the expressive power of query languages for matrices. In *ICDT*, pages 10:1–10:17, 2018.

[30] Trevor Campbell and Tamara Broderick. Automated scalable bayesian inference via hilbert coresets. *The Journal of Machine Learning Research*, 20(1):551–588, 2019.

[31] Timothy M. Chan, Kasper Green Larsen, and Mihai Pătraşcu. Orthogonal range searching on the ram, revisited. In *SoCG*, pages 1–10, 2011.

[32] Surajit Chaudhuri. Data mining and database systems: Where is the intersection? *IEEE Data Eng. Bull.*, 21(1):4–8, 1998.

[33] Surajit Chaudhuri, Usama M. Fayyad, and Jeff Bernhardt. Scalable classification over SQL databases. In *ICDE*, pages 470–479, 1999.

[34] Bernard Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988.

[35] Bernard Chazelle. Lower bounds for orthogonal range searching. II. The arithmetic model. *Journal of the ACM*, 37(3):439–463, 1990.

[36] Lingjiao Chen, Arun Kumar, Jeffrey F. Naughton, and Jignesh M. Patel. Towards linear algebra over normalized data. *PVLDB*, 10(11):1214–1225, 2017.

[37] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *KDD*, pages 785–794, 2016.

[38] Yu Cheng, Chengjie Qin, and Florin Rusu. Glade: Big data analytics made easy. In *SIGMOD*, pages 697–700, 2012.

[39] Philip A. Chou. Optimal partitioning for classification and regression trees. *IEEE Trans. Pattern Anal. Mach. Intell.*, 13(4):340–354, April 1991.

[40] C. Chow and C. Liu. Approximating discrete probability distributions with dependence trees. *IEEE Trans. Inf. Theor.*, 14(3):462–467, 2006.

[41] Ryan R. Curtin, Marcus Edel, Mikhail Lozhnikov, Yannis Mentekidis, Sumedh Ghaisas, and Shangtong Zhang. mlpack 3: A fast, flexible machine learning library. *J. Open Source Soft.*, 3:726, 2018.

[42] Ryan R. Curtin, Benjamin Moseley, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. Rk-means: Fast coreset construction for clustering relational data. To appear at AISTATS 2020, CoRR, abs/1910.04939, 2019.

[43] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational geometry*. Springer-Verlag, Berlin, third edition, 2008. Algorithms and applications.

[44] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *NIPS*, 2012.

[45] N.R. Draper and H. Smith. *Applied Regression Analysis*. Wiley Series in Probability and Statistics. Wiley, 2014.

[46] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, July 2011.

[47] Tarek Elgamal, Shangyu Luo, Matthias Boehm, Alexandre V. Evfimievski, Shirish Tatikonda, Berthold Reinwald, and Prithviraj Sen. SPOOF: sum-product optimization and operator fusion for large-scale machine learning. In *CIDR*, 2017.

[48] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, 2008.

[49] Corporacion Favorita. Corp. Favorita Grocery Sales Forecasting: Can you accurately predict sales for a large grocery chain?, `https://www.kaggle.com/c/favorita-grocery-sales-forecasting/`, 2017.

[50] Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. Towards a unified architecture for in-rdbms analytics. In *SIGMOD*, pages 325–336, 2012.

[51] Roger Fletcher. On the Barzilai-Borwein method. *Optimization and Control with Applications (Applied Optimization)*, 96:235–256, 2005.

[52] Zekai J. Gao, Shangyu Luo, Luis Leopoldo Perez, and Chris Jermaine. The BUDS language for distributed bayesian machine learning. In *SIGMOD*, pages 961–976, 2017.

[53] Georgios Giannikis, Darko Makreshanski, Gustavo Alonso, and Donald Kossmann. Shared workload optimization. *PVLDB*, 7(6):429–440, 2014.

[54] Steven G. Gilmour. The interpretation of mallows's c_p-statistic. *Journal of the Royal Statistical Society. Series D (The Statistician)*, 45(1):49–56, 1996.

[55] Tom Goldstein, Christoph Studer, and Richard G. Baraniuk. A field guide to forward-backward splitting with a FASTA implementation. *CoRR*, abs/1411.3406, 2014.

[56] Georg Gottlob, Gianluigi Greco, Nicola Leone, and Francesco Scarcello. Hypertree decompositions: Questions and answers. In *PODS*, pages 57–74, 2016.

[57] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing xpath queries. In *VLDB*, pages 95–106, 2002.

[58] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. In *PODS*, pages 21–32, 1999.

[59] Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *ICDE*, pages 152–159, 1996.

[60] Martin Grohe and Dániel Marx. Constraint solving via fractional edge covers. In *SODA*, pages 289–298, 2006.

[61] Torsten Grust. Accelerating xpath location steps. In *SIGMOD*, pages 109–120, 2002.

[62] Gaël Guennebaud, Benoît Jacob, Philip Avery, Abraham Bachrach, Sebastien Barthelemy, et al. Eigen v3, `http://eigen.tuxfamily.org`, 2010.

[63] S. Har-Peled and S. Mazumdar. On coresets for k-means and k-median clustering. In *SODA*, 2004.

[64] Sariel Har-Peled, Dan Roth, and Dav Zimak. Maximum margin coresets for active and noise tolerant learning. In *IJCAI*, pages 836–841, 2007.

[65] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. In *SIGMOD*, pages 205–216, 1996.

[66] T. Hastie, R. Tibshrani, and M. J. Wainwright. *Statistical Learning with Sparsity: The Lasso and generalizations*. CRC Press, 2015.

[67] Joseph M. Hellerstein, Christopher Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. The madlib analytics library or MAD skills, the SQL. *PVLDB*, 5(12):1700–1711, 2012.

[68] Botong Huang, Matthias Boehm, Yuanyuan Tian, Berthold Reinwald, Shirish Tatikonda, and Frederick R. Reiss. Resource elasticity for large-scale machine learning. In *SIGMOD*, pages 137–152, 2015.

[69] Laurent Hyafil and Ronald L. Rivest. Constructing optimal binary decision trees is NP-complete. *Information Processing Lett.*, 5(1):15–17, 1976/77.

[70] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[71] N. Ishakbeyoglu and Z. Ozsoyoglu. Testing satisfiability of a conjunction of inequalities. In *Int. Symposium on Computer and Inf. Syst. (ISCIS XII)*, pages 148–154, 1997.

[72] Anil K. Jain. Data clustering: 50 years beyond k-means. *Pattern Recognition Letters*, 31(8):651–666, 2010.

[73] Kaggle. The State of Data Science and Machine Learning, `https://www.kaggle.com/surveys/2017`, 2017.

[74] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *PVLDB*, 11(13):2209–2222, 2018.

[75] Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. What do shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? In *PODS*, pages 429–444, 2017.

[76] Benny Kimelfeld and Christopher Ré. A relational framework for classifier engineering. In *PODS*, pages 5–20, 2017.

[77] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *OOPSLA*, 1:77:1–77:29, 2017.

[78] Anthony C. Klug. On conjunctive queries containing inequalities. *Journal of the ACM*, 35(1):146–160, 1988.

[79] Arun Kumar, Jeffrey F. Naughton, and Jignesh M. Patel. Learning generalized linear models over normalized data. In *SIGMOD*, pages 1969–1984, 2015.

[80] Arun Kumar, Jeffrey F. Naughton, Jignesh M. Patel, and Xiaojin Zhu. To join or not to join? thinking twice about joins before feature selection. In *SIGMOD*, pages 19–34, 2016.

[81] Andrea Lincoln, Virginia Vassilevska Williams, Joshua R. Wang, and R. Ryan Williams. Deterministic time-space trade-offs for k-sum. In *ICALP*, pages 58:1–58:14, 2016.

[82] Stuart P. Lloyd. Least squares quantization in PCM. *IEEE Trans. Inf. Theory*, 28(2):129–136, 1982.

[83] Shangyu Luo, Zekai J. Gao, Michael N. Gubanov, Luis Leopoldo Perez, and Christopher M. Jermaine. Scalable linear algebra on a relational database system. *SIGMOD Rec.*, 47(1):24–31, 2018.

[84] Dániel Marx. Approximating fractional hypertree width. In *SODA*, pages 902–911, 2009.

[85] Dániel Marx. Tractable hypergraph properties for constraint satisfaction and conjunctive queries. *Journal of the ACM*, 60(6):Art. 42, 51, 2013.

[86] Wes McKinney. pandas: a foundational python library for data analysis and statistics. *Python for High Performance and Scientific Computing*, 14, 2011.

[87] H Brendan McMahan, Gary Holt, David Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, Eugene Davydov, Daniel Golovin, et al. Ad click prediction: A view from the trenches. In *KDD*, pages 1222–1230, 2013.

[88] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[89] Ronald Menich and Nik Vasiloglou. The future of LogicBlox machine learning. LogicBlox User Days, 2013.

[90] Jan Motl and Oliver Schulte. The ctu prague relational learning repository. *arXiv preprint arXiv:1511.03086*, 2015.

[91] Inderpal Singh Mumick, Dallan Quass, and Barinderpal Singh Mumick. Maintenance of data cubes and summary tables in a warehouse. In *SIGMOD*, pages 100–111, 1997.

[92] Kevin P. Murphy. *Machine learning: a probabilistic perspective*. MIT Press, Cambridge, Mass., 2013.

[93] Raghunath Othayoth Nambiar and Meikel Poess. The making of tpc-ds. In *PVLDB*, pages 1049–1058, 2006.

[94] John Ashworth Nelder and Robert WM Wedderburn. Generalized linear models. *Journal of the Royal Statistical Society: Series A (General)*, 135(3):370–384, 1972.

[95] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[96] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. In *PODS*, pages 37–48, 2012.

[97] Hung Q. Ngo, Christopher Ré, and Atri Rudra. Skew strikes back: New developments in the theory of join algorithms. In *SIGMOD Rec.*, pages 5–16, 2013.

[98] Dung T. Nguyen, Molham Aref, Martin Bravenboer, George Kollias, Hung Q. Ngo, Christopher Ré, and Atri Rudra. Join processing for graph patterns: An old dog with new tricks. *CoRR*, abs/1503.04169, 2015. Short version in GRADES@SIGMOD 2015.

[99] Dan Olteanu and Jiewen Huang. Secondary-storage confidence computation for conjunctive queries with inequalities. In *SIGMOD*, pages 389–402, 2009.

[100] Dan Olteanu and Maximilian Schleich. F: Regression models over factorized views. *PVLDB*, 9(10), 2016.

[101] Dan Olteanu and Maximilian Schleich. Factorized databases. *SIGMOD Rec.*, 45(2):5–16, 2016.

[102] Dan Olteanu and Jakub Závodný. Size bounds for factorised representations of query results. *TODS*, 40(1):2, 2015.

[103] Carlos Ordonez, Yiqun Zhang, and Wellington Cabrera. The gamma matrix to summarize dense and sparse data sets for big data analytics. *IEEE Trans. Knowl. Data Eng.*, 28(7):1905–1918, 2016.

[104] Mihai Patrascu and Ryan Williams. On the possibility of faster SAT algorithms. In *SODA*, pages 1065–1075, 2010.

[105] Judea Pearl. Reverend bayes on inference engines: A distributed hierarchical approach. In *AAAI*, pages 133–136, 1982.

[106] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent

Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. Scikit-learn: Machine learning in python. *J. Machine Learning Research*, 12:2825–2830, 2011.

[107] Jian Pei, Jiawei Han, and Laks VS Lakshmanan. Mining frequent itemsets with convertible constraints. In *ICDE*, pages 433–442, 2001.

[108] Holger Pirk, Oscar Moll, Matei Zaharia, and Samuel Madden. Voodoo - a vector algebra for portable database performance on modern hardware. *PVLDB*, 9:1707–1718, 2016.

[109] Chengjie Qin and Florin Rusu. Speculative approximations for terascale distributed gradient descent optimization. In *DanaC*, pages 1:1–1:10, 2015.

[110] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[111] J.R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann series in machine learning. Elsevier Science, 1993.

[112] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Stat. Comp., `www.r-project.org`, 2013.

[113] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701, 2011.

[114] Steffen Rendle. Factorization machines. In *ICDM*, pages 995–1000, 2010.

[115] Steffen Rendle. Factorization machines with libFM. *ACM Transactions on Intelligent Systems and Technology*, 3(3):57:1–57:22, 2012.

[116] Steffen Rendle. Scaling factorization machines to relational data. *PVLDB*, 6(5):337–348, 2013.

[117] C. Sanderson and R.R. Curtin. A user-friendly hybrid sparse matrix class in C++. In *Proceedings of the 2018 International Congrees on Mathematical Software (ICMS)*, pages 422–430. Springer, 2018.

[118] Maximilian Schleich, Dan Olteanu, Mahmoud Abo Khamis, Hung Q. Ngo, and XuanLong Nguyen. A layered aggregate engine for analytics workloads. In *SIGMOD*, pages 1642–1659, 2019.

[119] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. Learning linear regression models over factorized joins. In *SIGMOD*, pages 3–18, 2016.

[120] Gideon Schwarz. Estimating the dimension of a model. *The Annals of Statistics*, 6(2):461–464, 03 1978.

[121] Luc Segoufin. Enumerating with constant delay the answers to a query. In *ICDT*, pages 10–20, 2013.

[122] Timos K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.

[123] Amir Shaikhha, Yannis Klonatos, and Christoph Koch. Building efficient query engines in a high-level language. *ACM Trans. Database Syst.*, 43(1):4:1–4:45, 2018.

[124] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. How to architect a query compiler. In *SIGMOD*, pages 1907–1922, 2016.

[125] Amir Shaikhha, Maximilian Schleich, Alexandru Ghita, and Dan Olteanu. Multi-layer optimizations for end-to-end data analytics. In *CGO*, page 145–157, 2020.

[126] Shai Shalev-Shwartz, Yoram Singer, Nathan Srebro, and Andrew Cotter. Pegasos: primal estimated sub-gradient solver for svm. *Mathematical Programming*, 127(1):3–30, Mar 2011.

[127] Daniele G. Spampinato and Markus Püschel. A basic linear algebra compiler for structured matrices. In *CGO*, pages 117–127, 2016.

[128] Gilbert Strang. *Linear Algebra and Its Applications*. Thomson, Brooks/Cole, 2006.

[129] Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. *Probabilistic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

[130] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. How to architect a query compiler, revisited. In *SIGMOD*, pages 307–322, 2018.

[131] The StatsModels development team. StatsModels: Statistics in Python, `http://statsmodels.sourceforge.net`, 2012.

[132] Madeleine Udell, Corinne Horn, Reza Zadeh, and Stephen Boyd. Generalized low rank models. *Foundations and Trends in Machine Learning*, 9(1):1–118, 2016.

[133] Todd L. Veldhuizen. Leapfrog triejoin: A simple, worst-case optimal join algorithm. In *ICDT*, pages 96–106, 2014.

[134] Haizhou Wang and Mingzhou Song. Ckmeans.1d.dp: Optimal k-means clustering in one dimension by dynamic programming. *The R Journal*, 3(2):29, 2011.

[135] Abdul Wasay, Xinding Wei, Niv Dayan, and Stratos Idreos. Data canopy: Accelerating exploratory statistical analysis. In *SIGMOD*, pages 557–572, 2017.

[136] Hadley Wickham, Romain Francois, Lionel Henry, Kirill Müller, et al. dplyr: A grammar of data manipulation. *R package version 0.4*, 3, 2015.

[137] Weipeng P. Yan and Per-Åke Larson. Eager aggregation and lazy aggregation. In *VLDB*, pages 345–357, 1995.

[138] Yelp. Yelp dataset challenge, `https://www.yelp.com/dataset/challenge/`, 2017.

[139] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 2–2, 2012.

[140] Marcin Zukowski, Mark van de Wiel, and Peter Boncz. Vectorwise: A vectorized analytical dbms. In *ICDE*, pages 1349–1350, 2012.