University of Oxford

Learning Regression Models over Factorised Joins

Maximilian-Joel Schleich
Kellogg College

Prof. Dan Olteanu
Supervisor

Dissertation submitted in partial fulfilment of the degree of Master of Science in Computer Science, Department of Computer Science, University of Oxford

September 2015

Abstract

We investigate the problem of building least squares regression models over training datasets defined by arbitrary join queries on database tables. Our key observation is that joins entail a high degree of redundancy in both computation and data representation, which is not required for the end-to-end solution to learning over joins.

We propose a variant of batch gradient descent called **F** that can learn the parameters of a linear regression function in two passes over factorised representations of the datasets, even though they can be exponentially smaller than the flat relational datasets. We consider factorisations of asymptotically optimal sizes, which are governed by well-understood properties of the hypergraph of the join query Q.

Our system **F** exploits the factorisation structure, an algebraic rewriting of the regression's objective function f that decouples the computation of cofactors of model parameters in f from their convergence, and the commutativity of cofactor computation with relational union and projection.

Our approach enjoys both theoretical and practical properties. Given a database $\mathbf{D}$ and a join query Q, learning can be done in $O(|\mathbf{D}|^{\mathit{fhtw}(Q)})$ time, which is optimal for factorised join computation; in contrast, any relational engine can achieve at best $O(|\mathbf{D}|^{\rho^*(Q)})$, while there are classes of natural queries such as path, star, or hierarchical queries with the fractional edge cover number $\rho^*(Q)$ as large as the number of relations in the query and the fractional hypertree width $\mathit{fhtw}(Q)$ one. We experimented with real-world (MovieLens, LastFM, a large US retailer) and synthetic (house price market) datasets. We show that **F** outperforms R, which uses QR decomposition, and Python StatsModels, which uses Moore-Penrose pseudoinverse, by a factor that follows the compression ratio brought by the factorisation, which in our experiments is up to three orders of magnitude.

We further present an optimised version of **F**, called **F***, which is a stand-alone, end-to-end regression learner that takes any number of relations as input and outputs a fully learned regression model. **F*** pushes some of the computations performed by **F** into the factorised join algorithm. This leads to further performance improvements. **F*** can even take time less than computing the factorised join, which **F** requires as input.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Prof. Dan Olteanu for the constant guidance and the many hours we spent discussing the project. His support was a crucial factor of the success of this research project. I could not have asked for a better supervisor for my MSc project.

I would also like to thank Radu Ciucanu for participating in our discussions and his support in preparation of the experiments for the paper that we submitted.

My sincere thanks also goes to Dr. Tim Furche for the guidance throughout the year and for providing the machines to be able to run the experiments in this dissertation.

I am grateful to Alban Demiraj for encouraging me to apply to this course and the advice throughout the year.

Most importantly, I would like to thank my parents, my brother and my grandfather for their continuous support and encouragement. This year would not have been possible without them.

Contents

1 Introduction **1**

1.1 Motivation and Challenges . . . 2

1.2 Contributions . . . 2

1.3 Outline . . . 4

2 Preliminaries **5**

2.1 Factorised Databases: A Primer . . . 5

2.2 Linear Regression: A Primer . . . 8

2.2.1 Solving Least Squares Regression Problems . . . 12

3 Regression on Factorised Joins **15**

3.1 Rewriting the Parameter Cofactors . . . 20

3.2 Cofactor Computation over any Factorisation . . . 20

3.3 Extending F with non-linear basis functions . . . 23

3.4 F*: Pushing the F-layer into Factorised Join Computation . . . 25

3.4.1 Computing the F-layer in F* . . . 25

3.4.2 Strengths and Weaknesses of F* . . . 27

4 Implementation **29**

4.1 Implementation details for F . . . 29

4.1.1 Implementation of interaction terms in F . . . 30

4.2 Implementation details for F* . . . 30

4.3 Implementation of learning and convergence . . . 31

5 Experiments **33**

5.1 Experimental Setup . . . 34

5.1.1 Datasets and Learning Tasks . . . 34

5.2 Experimental Results . . . 36

5.2.1 Flat vs. Factorised Joins: Compression Ratio . . . 36

5.2.2 Flat vs. Factorised Joins: Performance . . . 37

5.2.3 Importing Datasets . . . 38

5.2.4 Learning without Feature Interactions 38
5.2.5 Learning with Feature Interactions 39
5.2.6 Breakdown of Learning Time for **F** 39
5.2.7 End-to-End Solution 39
5.2.8 Model selection 40
5.2.9 Comparing F and F* 40
5.2.10 Further findings 42

6 Related work **43**
6.1 Machine learning in databases 43
6.2 Gradient descent optimisation 44

7 Conclusions and Future Work **45**
7.1 Conclusions 45
7.2 Future Work 46
7.2.1 Extending F and F* 46
7.2.2 Extending the class of algorithms that benefit from factorisations 47

Bibliography **49**

Appendix **52**

Chapter 1

Introduction

The study of machine learning has received a lot of attention in academia as well as industry in recent years. This is largely due to the increasing reliance of major web-companies on machine learning techniques for their operations and projects. Notable examples are Google Brain, Facebook's DeepFace, or Amazon's recommender system. Machine learning is also prominently used is in retail and commercial analytics, for which there are three major approaches: descriptive, or backward-looking analytics, predictive, or forward-looking analytics such as classification and regression, and prescriptive analytics, which are also forward-looking and usually take the output of a predictive model as input. These commercial analytics models are built on typical retail data sources, such as weekly sales data, promotions, and product descriptions.

In this dissertation, the focus is on predictive analytics. In a retail setting, a typical prediction would estimate how much additional demand can be generated for a given product by a promotion. The goal is to create accurate predictions at a more granular level, so that it can be possible to do customer-specific promotions and separate out the impact of actions taken by the retailer (e.g., changing discounts and prices) from weather and environment [30].

In order to create these kind of models, it is necessary to include a wide range of data sources in the analytics to ensure that the model does not miss unknown patterns in the data. These data sources could be for instance: customer reviews, basket data transactions, competitive promotions and prices, flu trends, loyalty program history, customer transaction history, social media text related to the retailer products sold, store attributes such as demographics, weather, or nearby competition. However, adding more data sources implies that the load of data used for analytics significantly increases. Since machine learning techniques are computationally expensive to learn, this quickly leads to scalability issues.

In light of these limitations, it is common to manually partition the dataset into segments that current state-of-the-art commercial analytics systems can process independently [30]. The partitioning of data requires domain-expertise and since there is no unified approach it can be subjective. In a retail setting, a common approach is to split the data by markets or product categories. This approach, however, can lower the accuracy of the model, since it is possible that patterns that are present in the entire dataset are not accurately represented in the partitions. For example, segmenting by product categories means that unusual correlations between products will not be recognised (such as the well-know example of diapers and beer) and segmenting by markets implies that patterns in demand behaviour across different geographies cannot be leveraged. For this reason, the scalability issues for current state-of-the-art commercial analytics systems cause severe limitations to the preditive powers of the machine learning techniques applied.

1.1 Motivation and Challenges

In the commercial setting outlined above, the input data to these systems is usually relational, which implies that the data sources are stored in different tables. Machine learning algorithms, however, typically require a single design matrix that contains all the features that the algorithms learn on. For this reason, it is common practice to first join the input relations together and then to run analytics on the join output [30]. This approach is computationally expensive and a major limitation for real-time analytics.

The join also introduces redundant information in the input data for the analytics system. This additional information is not required for the machine learning algorithm to make accurate predictions and presents another limitation to the scalability of the analytics system.

An inspection of a comprehensive list of publicly-available commercial analytics systems has shown that for learning over joins they all have the inherent limitation of scanning the large flat join. Following this exploration and discussions with industry experts [30], three major shortcomings of the current state-of-the-art systems have been identified: (1) poor integration of analytics and databases, which are traditionally confined to distinct specialised systems in the ever-growing technology stack [3]; (2) poor efficiency already for few data sources; and (3) insufficient accuracy due to omission of further relevant data sources.

In order to overcome these limitations, there is increasing interest in academia and industry in building systems that integrate databases and machine learning [40, 2, 23]. The objective is to create a scalable system which at least allows for efficient joining of many relations and integrating analytics so that it can be run on large join results from inside the database.

1.2 Contributions

The contribution of this dissertation is a system, called **F**, that can build linear regression models on training datasets defined by arbitrary join queries on database relations. **F** exploits factorised representations of data to avoid redundancy in the input data. This means that the analytics system can effectively learn over the entire dataset instead of several partitions.

We extend the benefits of data factorisation to learning regression models, and show that the system presented in this dissertation only needs at most two passes over the factorised join to compute the model parameters.

The key observation underlying the proposed system is that the intermediate join step represents the main bottleneck and it is unnecessarily expensive. It entails a high degree of redundancy in both computation and data representation, yet this is not required for the end-to-end solution, whose result is a constant number of real-valued parameters of the learned model. By computing a *factorised join* [6] instead of the standard flat join, it is possible to reduce data redundancy and improve performance for both the join and the learning steps.

The factorised joins exploit properties of relational algebra, e.g., the distributivity of Cartesian product over union, to reduce data and computation redundancy. Intuitively, a join of several relations is a union of products of smaller relations and its factorisation avoids the materialisation of these products whenever possible.

The theoretical and practical gains in both required memory space and time performance for factorised joins are well-understood and can be asymptotically exponential in the size of the join [36]. Given a database $\mathbf{D}$ and a join query Q, $\mathbf{F}$ needs $O(|\mathbf{D}|^{\mathit{fhtw}(Q)})$ time, which is *worst-case optimal* for computing the factorised join; in contrast, any relational engine can achieve at best $O(|\mathbf{D}|^{\rho^*(Q)})$ [4, 34, 50]. There are classes of queries naturally occurring in learning-over-joins scenarios, e.g., path, star, hierarchical queries, where the fractional edge cover number $\rho^*(Q)$ can be as large as the number of relations in the query while the fractional hypertree width $\mathit{fhtw}(Q)$ is one. These theoretical guarantees for factorised joins translates to orders of magnitude for various datasets reported in experiments [6, 5].

Besides factorisation, $\mathbf{F}$ uses an algebraic rewriting of the regression model's objective function f that decouples the computation of the parameter cofactors, for which the dataset needs to be traversed, from the actual optimisation of the parameters. This separation of cofactor computation and optimisation implies that the process of optimising the parameters does not need to traverse the data anymore, so that we are able to learn regression models in only two passes over the factorised data. Cofactor computation commutes with relational union and projection, which enables further applications of $\mathbf{F}$. The commutativity with union allows us to compute the parameter cofactors for the entire dataset as the sum of corresponding cofactors for disjoint partitions, whose union make up the input dataset; this property is desirable for *concurrent learning*, where we can compute cofactors of partitions on different cores or machines. The commutativity with projection allows us to compute all cofactors once and then explore the space of learning functions by only running convergence for a subset of them; this property is desirable for *model selection*, whose goal is to find a subset of features that best predict a test dataset.

Experiments show that for a range of (public and synthetic) datasets, our approach can scale way beyond two popular open-source statistical software packages: **R** [39], which uses QR decomposition [20], dubbed as one of the ten most important algorithms of the 20th century[18], and **P**ython StatsModels [49], which uses Moore-Penrose pseudoinverse to compute closed-form solutions [37]. **F** needs orders-of-magnitude less memory and time to compute the parameters of the regression model with the same accuracy as its competitors. This experimental evidence is in tune with the theoretical guarantees mentioned above.

F and its worst-case bound is extended to learning linear regression functions over non-linear basis functions, which enables us to model arbitrarily complex, non-linear functions. This also includes feature interactions, which are used to understand relationships amongst features in the regression model.

We propose an optimisation of **F**, which we call **F***. Instead of computing the factorised join for input relations before learning, **F*** pushes some of the computations for learning over the join. This provides a performance speed-up but also restricts the flexibility of the analytics system.

Part of the research and experiments that were conducted for this dissertation, such as the explanation and analysis of **F**, has been submitted to the 2016 SIGMOD conference in [15].

1.3 Outline

This chapter has introduced the problem of limited scalability for current state-of-the-art commercial analytics systems. This problem prohibits the analytics systems to exploit the potential insight from available data and decreases the accuracy of the prediction. The remainder of the dissertation presents how we propose to tackle this scalability problem by exploiting factorised representations of data as well as the experimental evidence that shows that our approach can outperform current analytics systems significantly.

Chapter 2 provides an introduction to factorised data representations as well as linear regression. Section 2.2 also presents different ways to solve linear regression problems.

Chapter 3 will outline how linear regression models described in Chapter 2 can be learned in only two passes over factorised joins. The algebraic rewriting of the regression's objective function that is used to factor out the parameter cofactors is described in Section 3.1. Section 3.2 outlines the algorithm to compute the cofactors and explains how they are used to learn the regression model in two passes over the factorised data. Section 3.3 presents how the model can be extended to model non-linear functions. Section 3.4 explains how the regression learner can be optimised and provides the underlying algorithm for **F***. We also outline the strengths and limitations of **F***.

Chapter 4 provides more details on the actual implementation of our proposed system. This includes details on the implementation of **F** and **F***, the convergence algorithm used to optimise the proposed system, and more details on how interaction terms can be implemented.

Chapter 5 describes the datasets that we use for our experiments as well as the results of the experiments conducted to benchmark the proposed system.

Chapter 6 outlines the related work that addresses the scalability issues of commercial analytics systems as well as similar works in the area of scalable machine learning.

Chapter 7 concludes this dissertation and provides some areas for future work that extends the approach presented in this dissertation.

The **Appendix** contains proofs for prepositions that are stated in Chapter 3, as well as additional information on the datasets that are presented in Chapter 5.

Chapter 2

Preliminaries

This chapter describes the basic information about factorised databases and linear regression that is required to understand the remainder of the dissertation.

Factorised databases are the underlying framework for the join results that are exploited by the proposed system as input to learning regression models. The databases will be introduced by example; for a rigorous treatment, please refer to the literature [6, 5, 36].

Similarly for the introduction of linear regression, we focus on the parts that are required to understand this dissertation. More information on linear regression models can be found in [8] or [31].

2.1 Factorised Databases: A Primer

Factorised databases form a representation system for relational databases that exploits algebraic properties of relations, in particular the distributivity of the Cartesian product over union, to reduce data and computation redundancy.

To start with a simple example, consider a relation R over schema (A, B) that consists of a tuple for each combination of values $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$. Assuming the notation $\langle A : a\rangle$ for a singleton relation over schema A and with one tuple with value a, the relation R can be expressed in relational algebra as $\bigcup_{1\leq i,j\leq n} \langle A : a_i\rangle \times \langle B : b_j\rangle$. A possible factorisation of R is a product of two smaller relations: $R = R_A \times R_B$, where $R_A = \bigcup_{1\leq i\leq n} \langle A : a_i\rangle$ and $R_B = \bigcup_{1\leq j\leq n} \langle B : b_j\rangle$. This factorisation can naturally benefit aggregates. To count the tuples in R, we take the product of the counts of tuples in R_A and R_B. To sum over all A-values in R, we take the sum of all A-values in R_A and multiply with the count of tuples in R_B.

A particular application of this idea is represented by factorised joins [6]: Given a database $\mathbf{D}$ and an equi-join query Q, the result $Q(\mathbf{D})$ exhibits lots of (data and computation) redundancy that can be reduced by factorisation. Intuitively, a join of two relations is by definition a union of products of smaller relations: For every join value, several tuples from one relation can be paired with several tuples from the other relation. The factorised join avoids the materialisation of these products whenever possible.

Example 2.1 Figure 2.1(a) depicts a database consisting of three relations along with their natural join: The relation House records house prices and living areas (in squared meters) within locations given by zipcodes; TaxBand relates city/state tax

Shops

Z	H
z_1	h_1
z_1	h_2
z_1	h_3
z_2	h_4

House

Z	S	P
z_1	s_1	p_1
z_1	s_1	p_2
z_1	s_2	p_3
z_2	s_2	p_4

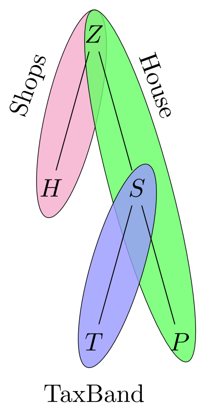
TaxBand

S	T
s_1	t_1
s_1	t_2
s_1	t_3
s_2	t_4
s_2	t_5

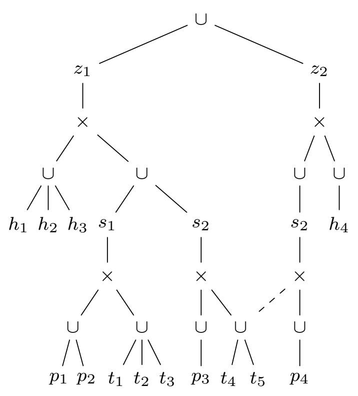
Shops ⋈ House ⋈ TaxBand

Z	H	S	P	T
z_1	h_1	s_1	p_1	t_1
z_1	h_1	s_1	p_1	t_2
z_1	h_1	s_1	p_1	t_3
z_1	h_1	s_1	p_2	t_1
z_1	h_1	s_1	p_2	t_2
z_1	h_1	s_1	p_2	t_3
z_1	h_1	s_2	p_3	t_4
z_1	h_1	s_2	p_3	t_5
………				
the above for h_2 and h_3				
………				
z_2	h_4	s_2	p_4	t_4
z_2	h_4	s_2	p_4	t_5

(a) Relations of database $\mathbf{D}$ and natural join $Q(\mathbf{D})$.



(b) F-tree F.



(c) Factorisation $F(\mathbf{D})$.

Figure 2.1: (a) Database $\mathbf{D}$ with relations House(Zipcode, Sqm, Price), TaxBand(Sqm, Tax), Shops(Zipcode, Hours), where the attribute names are abbreviated and the values are not necessarily distinct; (b) Nesting structure (f-tree) for the natural join of the relations; (c) Factorisation $F(\mathbf{D})$ of the natural join over F.

bands with house living areas; Shops list shops with zipcode and opening hours (for reasons of brevity, we omit further relevant attributes; in our experiments, we consider a real dataset from a large US retailer that is an extension of this scenario).

The join result exhibits a high degree of redundancy. For instance, the value z_1 occurs in 24 tuples, each value h_1 to h_3 occurs in eight tuples and they are paired with the same combinations of values for the other attributes. Since z_1 is paired in relation House with p_1 to p_3 and in relation Shops with h_1 to h_3, all combinations (indeed, the Cartesian product) of the former and the latter values occur in the join result. We can represent this local product symbolically instead of eagerly materialising it. If we systematically apply this observation, we obtain an equivalent factorised representation of the entire join result that is much more compact than the flat, tabular representation of the join result.

For instance, consider the first three tuples from the join result in Figure 2.1(a) expressed as a relational algebra expression:

$$\begin{aligned}\phi = &\langle Z : z_1\rangle \times \langle H : h_1\rangle \times \langle S : s_1\rangle \times \langle P : p_1\rangle \times \langle T : t_1\rangle\\ \cup &\langle Z : z_1\rangle \times \langle H : h_1\rangle \times \langle S : s_1\rangle \times \langle P : p_1\rangle \times \langle T : t_2\rangle\\ \cup &\langle Z : z_1\rangle \times \langle H : h_1\rangle \times \langle S : s_1\rangle \times \langle P : p_1\rangle \times \langle T : t_3\rangle.\end{aligned}$$

By exploiting the distributivity of the Cartesian product over union, the factorisation of the same three tuples in the join result would be:

$$\phi = \langle Z : z_1\rangle \times \langle H : h_1\rangle \times \langle S : s_1\rangle \times \langle P : p_1\rangle \times \big(\langle T : t_1\rangle \cup \langle T : t_2\rangle \cup \langle T : t_3\rangle\big).$$

Figure 2.1(c) shows a factorisation of the entire join result (we dropped the attribute names from singletons as they are clear from context). Each tuple in the join result is represented once in the factorisation and can be constructed by following one branch of every union and all branches of a product. To count the number of represented tuples, we take the sum (product) of the counts of children for each union (respectively, product) and count each singleton as one. □

The factorised join in Figure 2.1(c) has the nesting structure depicted in Figure 2.1(b): It is a union of Z-singletons occurring in both Shops and House relations, i.e., in their join on Z. For each Z-singleton z, we represent separately the union of H-singletons paired with z in relation Shops and the union of S-singletons paired with z in relation House and with T-singletons in TaxBand. In other words, given z, the H-singletons are independent of the S-singletons and can be stored separately; this is where the factorisation saves computation and space as it avoids an explicit enumeration of all combinations of H-singletons with S-singletons for a given z. Furthermore, under each S-singleton, there is a union of T-singletons and a union of P-singletons.

Such nesting structures are called *factorisation trees*, or f-trees for short. They are depicted as trees, where the nodes are attributes, cf. Figure 2.1(b). The f-trees satisfy the *path constraint* that all attributes of a relation lie along the same root-to-leaf path, since they are not independent in general [36]. Attributes from different relations are independent and may lie on different branches, e.g., H is independent of all attributes but Z. The f-trees represent partial orders of the equi-join conditions in the query and as such can be derived from the query. In practice, their construction is guided by cardinality and join selectivity estimates.

The factorisation can be further compacted by *caching* common subexpressions [36]. Similarly to f-trees, caching can be statically inferred from the join query (or the instance). In our example, a given S-singleton s_2 occurs with its union of T-singletons $t_4 \cup t_5$, regardless of which Z-singletons s_2 is paired with. We can therefore represent this union once and reuse it for every occurrence of s_2.

F-trees can lead to factorisations of greatly varying sizes, where the size of a representation (flat or factorised) is defined as the number of its singletons. Within the class of factorisations over f-trees, we can find the worst-case optimal ones and also compute them in worst-case optimal time:

Proposition 2.2 *Given a join query Q, for any database $\mathbf{D}$, the join result $Q(\mathbf{D})$ admits*

- *a flat representation of size* $\Theta(|\mathbf{D}|^{\rho^*(Q)})$ [4]*;*
- *a factorisation without caching of size* $\Theta(|\mathbf{D}|^{s(Q)})$ [36]*;*
- *a factorisation with caching of size* $\Theta(|\mathbf{D}|^{\mathit{fhtw}(Q)})$ [36]*.*

There are worst-case optimal join algorithms to compute the join result in each of the three representations [34, 36]*.*

The measures used in Proposition 2.2 are: the fractional edge cover number $\rho^*(Q)$, the factorisation number $s(Q)$, and the fractional hypertree width $\mathit{fhtw}(Q)$. We know that $1 \leq \mathit{fhtw}(Q) \leq s(Q) \leq \rho^*(Q) \leq |Q|$, where $|Q|$ is the size (number of relations) in query Q [36]. For large classes of join queries, e.g., for hierarchical (including star) queries, $s(Q) = 1$ while $\rho^*(Q) = |Q|$. Moreover, the gap between $\mathit{fhtw}(Q)$ and $s(Q)$ can be as much as $\log |Q|$; it is $\log |Q|$ for path queries where $\mathit{fhtw}(Q) = 1$. Clique queries (e.g., triangles) are the pathological cases for which factorisations bring no asymptotic saving. The fractional hypertree width is fundamental to problem tractability with applications spanning constraint satisfaction, databases, matrix operations, probabilistic graphical models, and logic [26].

Example 2.3 We first consider the flat representation of the natural join Q in our example. A trivial upper bound is the product of the sizes of the input relations, so $O(|\mathbf{D}|^3)$ assuming all input relations have size $|\mathbf{D}|$. We can construct a lower bound that matches the upper bound for a class of databases where the Z and S values are the same across all tuples and the attributes H, T, and P have as many distinct values as $|\mathbf{D}|$; in this case, the join is essentially a product of the three sets of values for H, T, and P. This means that the fractional edge cover number governing the size of the flat join is three in our example: $\rho^*(Q) = 3$.

We next consider the factorised join without caching. The above argument for the lower bound for the flat join would yield a size quadratic in $|\mathbf{D}|$. The factorisation for the branch $Z - H$ in the f-tree has size at most linear in $|\mathbf{D}|$, since for a given Z-singleton we list the union of its H-singletons and their overall number is bounded by the number of singletons in relation Shops. The size of the factorisation for this branch is independent of the size of the branch $Z - S$, since the singletons for H and S are represented independently of each other. The number of singletons for S and P is bounded by the number of of singletons in relation House. The number of T-singletons can however be quadratic in $|\mathbf{D}|$: This is the case when we have one S-singleton paired with $|\mathbf{D}|$ T-singletons in TaxBand and with $|\mathbf{D}|$ Z-singletons in House. This means that the factorisation number in our example is two: $s(Q) = 2$.

We finally consider the factorised join with caching. In contrast to the previous case, the construction used to attain the quadratic lower bound does not work anymore: We cache the union of T-singletons for the S-singleton and reuse it under every Z-singleton. This means that the fractional hypertree width in our example is one: *fhtw*$(Q) = 1$. $\square$

2.2 Linear Regression: A Primer

Linear Regression is one of the central algorithms in the realm of supervised learning. In a typical linear regression problem, one is given a training dataset of size m that consists of input-output pairs.

$$\{(y^{(1)}, x_1^{(1)}, \ldots, x_n^{(1)}), \ldots, (y^{(m)}, x_1^{(m)}, \ldots, x_n^{(m)})\}.$$

Inputs $(x_j^{(i)})$ are commonly referred to as predictors or features; while outputs $(y^{(1)})$ are known as targets or labels. We also define the features and labels in matrix notation as follows:

$$\mathbf{y} = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{pmatrix}, \ \mathbf{X} = \begin{pmatrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(m)} & x_2^{(m)} & \cdots & x_n^{(m)} \end{pmatrix}$$

Additionally, we define $\mathbf{x}^{(i)}$ to be one feature vector $\mathbf{x}^{(i)} = (x_1^{(i)}, \ldots, x_n^{(i)})$.

In machine learning, the training dataset is also often referred to as the *design matrix* for the learning algorithm. If the input data is relational, then the dataset is computed by taking the join of all input relations. For simplicity, it is assumed in this dissertation that all features and labels are real numbers. The goal of linear

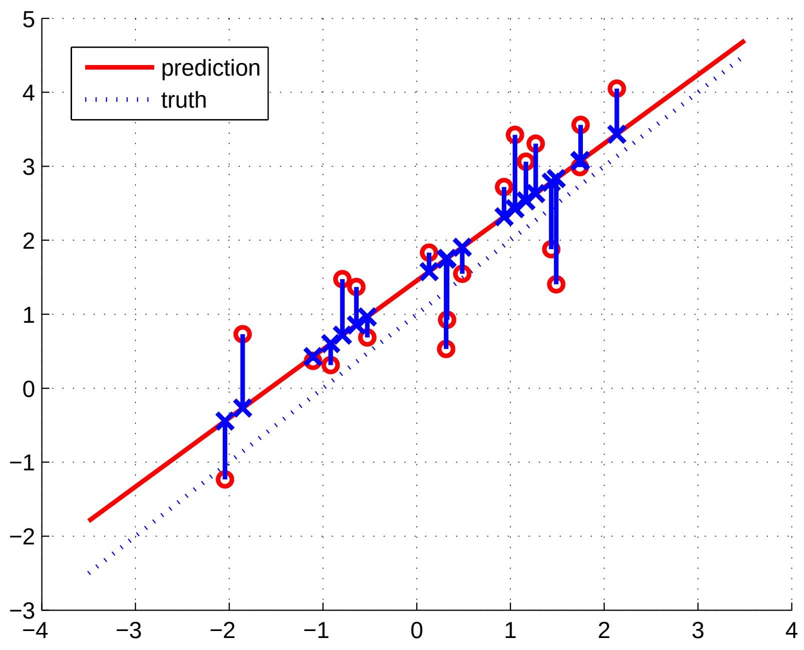


Figure 2.2: Example of a simple one-dimensional least squares regression problem. The goal is to minimise the sum of squared errors (visualised by the blue lines) so that the model (indicated by the red line) best describes the data provided (red points). Figure taken from [31].

regression is to learn a set of parameters $\theta = (\theta_0, \ldots, \theta_n)^T$ so that the linear function

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \ldots + \theta_n x_n$$

accurately predicts the labels for new, unknown features. It is common to include $x_0 = 1$ in the data so that $h_\theta(x) = \sum_{k=0}^{n} \theta_k x_k$. This is equivalent to adding a column of ones to the feature matrix $\mathbf{X}$.

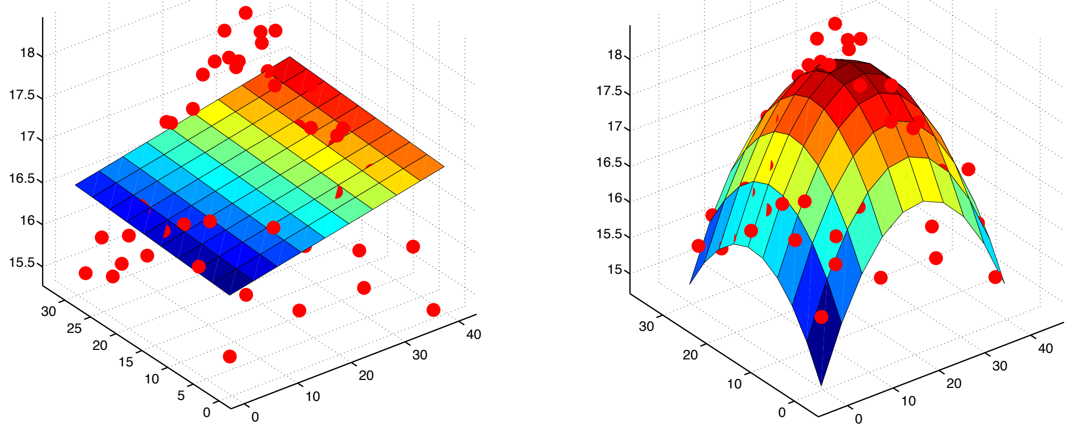
For the training dataset provided in Figure 2.1(a), it is natural to use the available features to predict the price of the house, which implies that the label would be P.

In order to determine how well the model predicts the label, it is common to use the so-called least squares regression objective function

$$\begin{aligned} J(\theta) &= \frac{1}{2}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 \\ &= \frac{1}{2}(\mathbf{X}\theta - \mathbf{y})^T(\mathbf{X}\theta - \mathbf{y}) \qquad (2.1) \end{aligned}$$

This function measures the error of the model, which is measured by how far the predictions $(h_\theta(x))$ are away from the actual values (y). The goal of the optimisation procedure is to find the set of parameters that minimises the objective function.

Example 2.4 Figure 2.2 provides a visual representation of the problem that a least squares regression model aims to solve. The red points in the figure indicate the



(a) Identity Basis Function (b) Polynomial Basis Function

Figure 2.3: Comparison of linear regression problem with two-dimensional inputs where the features are defined by the identity basis function in (a) and polynomial basis function with degree $d = 2$ in (b). Figure taken from [31].

location of the samples or input-output data pairs. The x-axis indicates the one-dimensional feature $\mathbf{x} = (1, x_1)$ and the y-axis gives the actual label y for the given feature $\mathbf{x}$. The red line indicates the model that we are aiming to optimise: $h_\theta(x) = \theta_0(1) + \theta_1(x_1)$. The blue crosses along the red line indicate the prediction of the model for the given input features. The blue line between the prediction $h_\theta(x)$ and the true label y indicate the residual, or error, of the model. The goal of least squares regression is to minimise the sum of squared errors, which is visualised by finding the line that minimises the total length of the blue residual lines. $\square$

It can be shown that least squares regression is equivalent to a model that uses gaussian probability distributions to estimate the noise. This would then be optimised by finding the parameters that maximise the likelihood of fitting this model to the given training data.

Although, linear regression is inherently linear with respect to the parameters, it is possible to model complex, non-linear functions by replacing the input feature $\mathbf{x} = (x_1, \ldots, x_n)$ by a non-linear basis function $\phi(\mathbf{x}) = (\phi_1(x_1), \ldots, \phi_n(x_n))$. Simple basis functions are the polynomial basis functions, which, for a single feature x, add the polynomial terms up to a degree d to the feature space. This means that the polynomial basis function for x has the form: $\phi(x) = (x, x^2, x^3, \ldots, x^d)$. Increasing d will make the function, which is predicted by the model, increasingly complex.

Example 2.5 Figure 2.3 gives an example of two-dimensional input data with two different basis functions: (a) the identity basis function and (b) the polynomial basis function with degree $d = 2$. The use of different basis functions results in two different models, which are visualised by the surface plots in the two images. For the identity basis function, the model is given by a plane of the form: $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2$. For the polynomial basis function, on the other hand, the model is defined by a quadratic function of the form: $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2$. $\square$

Other common basis functions are the Gaussian Radial or the Sigmoidal Basis Functions [8]. The former has the form:

$$\phi_j(x) = \exp\left\{-\frac{(x-\mu_j)^2}{2s^2}\right\},$$

where μ and s are parameters that determine the location and scale of the basis function. They follow the shape of a Gaussian probability distribution but they do not necessarily have a probabilistic interpretation.

The Sigmoidal Basis Function is defined as:

$$\phi_j(x) = \sigma\left(\frac{(x-\mu_j)}{s}\right),$$

where $\sigma(a) = \frac{1}{1+e^{-a}}$ is the sigmoid function. All of these functions add non-linearity to the model so that arbitrarily complex functions can be approximated.

Although basis functions can be used to model increasingly complex functions, they have to be applied with caution. If the model is too complex it leads to overfitting, where the model perfectly fits the training data but does not resemble the true underlying function. As a result the model would have poor performance when it is tested on new data after training.

In addition to adding more complexity to the model, basis functions can also be used to understand more about the features in a model. In vanilla linear regression, a feature is used to predict the value of the label. If several features are provided, then we predict the label based on the weighted sum of these features. This assumes that the features are mutually independent, which is a strong assumption to make. It could be possible that the prediction is not affect by the addition of the features but rather by the interaction of the features. In order to capture this information, it is common to include interaction terms whenever two features are not independent [24].

Interaction terms are constructed from the original feature space. If a model contains two features x_1 and x_2 then the linear model would be of the following form: $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2$. When the features are not independent, however, the interaction between features x_1 and x_2 is modelled by the product of the features:

$$\phi(x_i, x_j) = x_i \cdot x_j.$$

This would result in the following model:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1 x_2$$

Interaction terms change the interpretation of the parameter. In standard linear regression, the parameter of a feature shows how much the prediction would change if the value of one feature would increase by one unit, we call this the effect of the feature. The inclusion of the interaction term implies that the the effect of one feature is conditionally dependent on the other feature. This can be shown by rewriting the model from above as follows:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + (\theta_2 + \theta_3 x_1) x_2.$$

The parameter of x_2 is given by $\theta_2 + \theta_3 x_1$, which implies that a unit increase in the value of x_1 changes the effect that x_2 has on the prediction by θ_3.

If several features in the model are not mutually independent, then pairwise interaction terms are added for each interaction effect. In order to ensure that the model can be interpreted it is necessary that the features that go into an interaction term are also individually included in the model.

2.2.1 Solving Least Squares Regression Problems

By differentiating the objective function, it is possible to find the closed form solution to the least squares regression problem. This is also known as the ordinary least squares solution.

$$\theta = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$$

This approach has the advantage that the parameters of the model are calculated exactly. On the other hand, it is computationally expensive because it requires the computation of the inverse of the design matrix which takes $O(m^3)$ time. For this reason, it is now common practice to use optimisations that reduce the computational cost of this closed-form solution. One such approach is using the QR-decomposition [20] to avoid calculating the inverse of the complete design matrix. We will give a short introduction to QR decomposition, for a more in-depth analysis please refer to the literature [47, 31, 48].

The QR decomposition can be used to express a $m \times n$ matrix $\mathbf{A}$ as a product of one $m \times m$ orthogonal matrix $\mathbf{Q}$ and one $n \times n$ upper triangular matrix $\mathbf{R}$. For simplicity we will assume that $\mathbf{A}$ is full rank, if this is not the case the algorithm requires a slight modification, for details please refer to [47].

The decomposition into $\mathbf{Q}$ and $\mathbf{R}$ is beneficial because orthogonal matrices have the property that $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}_m$. This property also implies that the inverse $\mathbf{Q}^{-1}$ is identical to the transpose $\mathbf{Q}^T$.

The underlying procedure to find the QR decomposition relies on an orthogonalisation algorithm, called Gram-Schmidt procedure [48], which orthogonalises a sequence of vectors by subtracting the projection of the previously orthogonalised vectors from each vector. The orthogonalised vectors are then normalised to have normal one. This procedure creates the orthnormal vectors q_i, which form the columns of the $\mathbf{Q}$ matrix from the QR decomposition. The additional matrix $\mathbf{R}$ connects the orthonormal columns in $\mathbf{Q}$ to the original matrix $\mathbf{A}$.

For least squares regression problems, the QR decomposition is applied to decompose the design matrix $\mathbf{X}$ and to exploit the properties of orthogonal matrices to compute the closed form solution in a more cost efficient way. We now show how the decomposition helps to compute the solution to the least squares problem:

Using the QR decomposition, let $\mathbf{X} = \mathbf{QR}$, where $\mathbf{Q}$ is orthonormal and $\mathbf{R}$ is upper triangular. Given that for an orthonormal matrix $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}_m$, we can rewrite $(\mathbf{X}^T\mathbf{X})^{-1}$ as follows:

$$(\mathbf{X}^T\mathbf{X})^{-1} = (\mathbf{R}^T\mathbf{Q}^T\mathbf{QR})^{-1} = (\mathbf{R}^T\mathbf{R})^{-1} = \mathbf{R}^{-1}\mathbf{R}^{-T}$$

Therefore, we can reformulate the ordinary least squares solution as follows:

$$\begin{aligned}\theta &= (\mathbf{X}^T\,\mathbf{X})^{-1}\,\mathbf{X}^T\,\mathbf{y} \\ &= \mathbf{R}^{-1}\mathbf{R}^{-T}\mathbf{R}^T\mathbf{Q}\mathbf{y} \\ &= \mathbf{R}^{-1}\mathbf{Q}\mathbf{y}\end{aligned}$$

This reformulation is advantageous, because it replaces the expensive computation of $(\mathbf{X}^T\mathbf{X})^{-1}$ by computing the inverse of $\mathbf{R}$, which is computationally inexpensive since $\mathbf{R}$ is an upper triangular matrix.

The QR-decomposition of an $m \times n$ matrix can be computed in $O(mn^2)$ time. The QR decomposition is typically the method of choice for solving least squares regression problems [31].

As an alternative to the closed-form solution, it is possible to approximate the parameters with iterative optimisation methods. The underlying method that is used for the proposed system of this dissertation is *batch gradient descent* [8], which repeatedly updates the parameters of h_θ in the direction of the gradient to decrease the error given by $J(\theta)$.

$$\begin{aligned}\forall 0 \leq j \leq n : \theta_j &:= \theta_j - \alpha \frac{\delta}{\delta\theta_j} J(\theta) \\ &:= \theta_j - \alpha \sum_{i=1}^{m} (\sum_{k=0}^{n} \theta_k x_k^{(i)} - y^{(i)}) x_j^{(i)}.\end{aligned}$$

In the above expression, α is called the learning rate, which regulates the size of the step by which the parameters are updated in the direction of the gradient. The learning rate has to be chosen with care, because if it is too small, the optimisation procedure takes too long to converge; if is too large, the algorithm will start to oscillate around the solution but not converge towards it. For this reason, it is common to let the learning rate adapt with the number of conversion steps. A popular method for adaptive learning rates is called AdaGrad [19], which is described in detail in Section 4.3.

A naïve implementation of the batch gradient descent algorithm would start with some initial values of parameters θ_k and perform one pass over the dataset to compute the value of the gradient, followed by one approximation step for the parameters, and repeating this process until convergence. Example 2.6 provides a visual representation of the iterative optimisation process and the effect of the learning rate.

Example 2.6 Figure 2.4 shows the contour plot for an exemplary optimisation problem with a global minimum at point $(1, 1)$. Assuming the batch gradient descent algorithm starts at point $(0, 0)$, then the red lines show the first 20 optimisations steps for learning rates $\alpha = 0.1$ and $\alpha = 0.6$, in (a) and (b) respectively.

The two plots show the tradeoff that needs to be considered when choosing the learning rate for optimisation. When the learning rate is too small, the algorithm will take a long time to converge (as shown in (a)) but if it is too big the algorithm will take large steps at the beginning and then begin to oscillate around the optimum. This is shown in (b). □

Batch gradient descent optimisation is not used in practice since it is inefficient to go over the entire dataset for each iteration. In this dissertation, however, it is

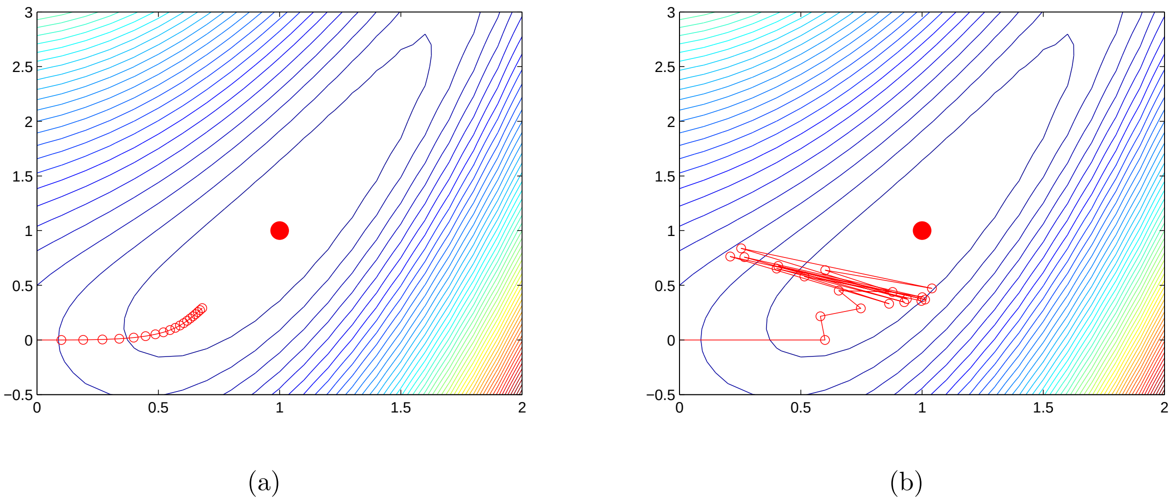


Figure 2.4: Batch gradient descent steps in an optimisation problem with learning rate (a) $\alpha = 0.1$ and (b) $\alpha = 0.6$. Figure taken from [31].

shown that batch gradient descent can be very competitive when adapted to work on succinct factorised relations.

Chapter 3

Regression on Factorised Joins

This chapter presents how the proposed system **F** benefits from an algebraic rewriting of the objective function to compute linear regression models in just two passes over any given factorised join. In Section 3.4, we then introduce an optimised version, called **F***, which pushes some of the computations performed by **F** into the computation of the factorised join.

The bulk of the computation for learning with batch gradient descent is done during the optimisation of the parameters, which is based on the gradient of the objective function. Therefore, we consider the following function for our algebraic rewriting:

$$\forall\, 0 \leq j \leq n : \theta_j := \theta_j - \alpha \sum_{i=1}^{m} (\sum_{k=0}^{n} \theta_k x_k^{(i)} - y^{(i)}) x_j^{(i)}. \tag{3.1}$$

In an naïve approach, the gradient is computed at every single iteration, which implies that the entire dataset is traversed at each convergence step.

The key insights of the proposed system rely on the observation that the batch gradient descent algorithm has two logically independent computation tasks that are intertwined in the above expression: the computation of the gradient and the convergence of the parameters.

The gradient expression can be considered a simple sum aggregate over the data. Let S_j be the sum aggregate for parameter j, which is equivalent to the gradient for j. The sum aggregate can be simplified without loss of generality by assigning a predefined parameter $\theta_y = -1$ to the label y. This means that the label can be considered as part of the input features:

$$\forall\, 0 \leq j \leq n : S_j = \sum_{i=1}^{m} (\sum_{k=0}^{n+1} \theta_k x_k^{(i)}) x_j^{(i)} \tag{3.2}$$

This approach is guided by two main insights, which are treated in more depth in the next sections.

Our **first insight** is that for a given parameter θ_k the sum aggregate S_j reduces to $\theta_k \times \sum_{i=1}^{m}(x_k^{(i)} x_j^{(i)})$. We define this rewritten expression as the multiplication between the parameter θ_k and the corresponding *cofactor* for parameter θ_k and sum aggregate S_j: Cofactor$[k, j]$. This allows for the following rewriting of the sum aggregate from Equation (3.2):

$$\forall 0 \leq j \leq n : S_j = \sum_{k=0}^{n+1} \theta_k \times \text{Cofactor}[k, j] \tag{3.3}$$

Since the cofactors remain the same for each convergence iteration, it is possible to separate the computation of the cofactors from parameter optimisation. This is crucial for performance as it mitigates the problem of scanning the entire dataset at each convergence step. The proposed system $\mathbf{F}$ computes the cofactors once and performs parameter convergence directly on the matrix of cofactors, whose size is independent of the data size m.

Furthermore, the cofactor matrix has desirable properties, which are presented in Proposition 3.1. The proofs are provided in the Appendix.

Proposition 3.1 *Let* $(Q, \mathbf{D})$ *be a pair of join query* Q *and database* $\mathbf{D}$ *defining the training dataset* $Q(\mathbf{D})$ *with schema/features* $\sigma = (A_0, \ldots, A_n)$. *Let* Cofactor *be the cofactor matrix for learning the parameters* $\theta_{A_0}, \ldots, \theta_{A_n}$ *of the function* $f_\theta = \sum_{k=0}^{n}(\theta_{A_k} x_{A_k})$ *using batch gradient descent.*

The cofactor matrix has the following properties:

1. Cofactor *is symmetric:*

$$\forall\, 0 \leq k, j \leq n : \text{Cofactor}[A_k, A_j] = \text{Cofactor}[A_j, A_k].$$

2. Cofactor *computation commutes with union: Assume* $Q(\mathbf{D}_1), \ldots, Q(\mathbf{D}_p)$ *are training datasets with cofactor matrices* $\text{Cofactor}_1, \ldots,$ Cofactor_p *where* $\mathbf{D} = \bigcup_{l=1}^{p} \mathbf{D}_l$, *then*

$$\forall\, 0 \leq k, j \leq n : \text{Cofactor}[A_k, A_j] = \sum_{l=1}^{p} \text{Cofactor}_l[A_k, A_j].$$

3. Cofactor *computation commutes with projection: Given a feature set* $L \subseteq \sigma$ *and the cofactor matrix* Cofactor_L *for the training dataset* $\pi_L(Q(\mathbf{D}))$, *then*

$$\begin{aligned} &\forall\, 0 \leq k, j \leq n \text{ such that } A_k, A_j \in L : \\ &\qquad \text{Cofactor}_L[A_k, A_j] = \text{Cofactor}[A_k, A_j]. \end{aligned}$$

The symmetry property implies that it suffices to only compute the upper half of the cofactor matrix.

Cofactor computation commutes with union in the sense that the cofactor matrix for the union of several training datasets is the entry-wise sum of the cofactor matrices of these training datasets. This commutativity property is key to the efficiency of our approach, since we can compute partial cofactors locally and add up cofactors over different partitions of the training dataset. It is also desirable for *concurrent computation*, because it implies that cofactors of different partitions can be computed on different cores or machines.

The commutativity with projection means that the cofactor matrix can be used to compute any subset of the parameters: All it takes is to ignore from the matrix the columns and rows for the parameters that are not considered for the learning problem. This is beneficial because there can be attributes in the dataset that are irrelevant for learning but relevant for constructing the join output. Prime examples are identifiers such as the relation keys supporting the join, e.g., zipcode in the training dataset in Figure 2.1(a). It is also beneficial for *model selection*, a key challenge in machine learning centred around finding the subset of features that best predict a test dataset. Model selection is a laborious and time-intensive process, since it requires to learn independently parameters corresponding to subsets of the available features. By decoupling the cofactor computation from parameter optimisation, it is possible to first compute the cofactor matrix for all features and then perform convergence on top of the cofactor matrix for the entire lattice of parameters independently of the data. Besides choosing the features after cofactor computation, it is also possible to use different attributes as labels by fixing its model parameters to -1. This implies that the proposed system allows for learning multiple predictions efficiently.

The two commutativity properties in Proposition 3.1 hold under bag (SQL) semantics in the sense that the relational projection and union operators do not remove duplicates. This is important, since learning is sensitive to duplicates.

The **second insight** is that cofactors can be computed in two passes over *any factorised join* representing the training dataset, which has the flat join as a special case:

Proposition 3.2 *Let $(Q, \mathbf{D}, F)$ be a triple of a join query Q, database $\mathbf{D}$, and any f-tree F of Q. Let the training dataset be the factorised join $F(\mathbf{D})$ over the f-tree F with attributes $\sigma = (A_0, \ldots, A_n)$, and let* Cofactor *be the cofactor matrix for learning the parameters $\theta_{A_0}, \ldots, \theta_{A_n}$ of the function $f_\theta = \sum_{k=0}^{n}(\theta_{A_k} x_{A_k})$ using batch gradient descent.*

Then, Cofactor *can be computed in two passes over the factorised join $F(\mathbf{D})$.*

Section 3.2 gives algorithms to compute the cofactor matrix. An immediate implication is that the redundancy in the flat join result is not necessary for learning:

Theorem 3.3 *The parameters of any linear function over features from a training dataset defined by a database $\mathbf{D}$ and a join query Q can be learned in time $O(|\mathbf{D}|^{fhtw(Q)})$.*

Theorem 3.3 is a direct corollary of Propositions 2.2 and 3.2. We recall our discussion in Section 2.1 that within the class of factorised representations with caching whose nesting structures are defined by joins, this time complexity is essentially worst-case optimal in the sense that there is no join algorithm that can achieve a lower worst-case time complexity. To put this result into a broader context, any worst-case optimal join algorithm that would produce flat relational results, such as NPRR [34] or LogicBlox's LeapFrog TrieJoin [50], would need time at least $O(|\mathbf{D}|^{\rho^*(Q)})$ to create the training dataset, yet the gap between $\rho^*(Q)$ and $fhtw(Q)$ can be as large as the number of relations in the join query.

The **two insights** discussed above complement each other; in particular Proposition 3.1 still holds in the presence of factorised joins. The commutativity with projection is especially useful in conjunction with factorisation since it does not restrict our choice of possible f-tree for the factorised join depending on the input features used for learning. In order to find the best factorisation it may be beneficial to include attributes (e.g. join attributes) which are not relevant for learning and can be skipped over at learning time. Explicitly removing attributes from the factorised join may in fact lead to larger representations (this contrasts with the flat case). For instance, if one would eliminate from the factorised join in Figure 2.1(c) all singletons for attributes Z and S, then the remaining attributes H, T, and P would become dependent on each other (they were independent conditioned on values for Z and S). The only permissible f-trees would be paths, and the factorised join may be asymptotically as large as the flat join.

The discussed approach can be extended with any of the non-linear basis functions that were discussed in Section 2.2. We will provide a detailed description of how basis functions can be incorporated in **F** in Section 3.3.

In the next sections, we discuss the properties of the cofactor matrix and then show how to compute it. The arithmetic expressions defining the cofactors in Equation (3.3) can be written much more compactly to avoid a great deal of redundancy. Remarkably, these rewritings are already performed in the factorised join, and, as shown in Section 3.2, the rewritten cofactors can be calculated with two passes over the factorised join.

There exist two possible rewritings for calculating the cofactors, which depend on whether the two features X, Y for Cofactor$[X, Y]$ come from the same input relation. The arithmetic rewritings will be explained by means of two examples which outline the differences between the two cases.

Example 3.4 First, inspect the sum aggregates in Equation (3.2) for the training dataset TD from Figure 2.1(a). There is one sum aggregate per attribute or feature column in the dataset. For attribute Z, we obtain:

$$\begin{aligned}
S_Z = &(\theta_Z z_1 + \theta_H h_1 + \theta_S s_1 + \theta_P p_1 + \theta_T t_1)z_1 + \\
&(\theta_Z z_1 + \theta_H h_1 + \theta_S s_1 + \theta_P p_1 + \theta_T t_2)z_1 + \\
&(\theta_Z z_1 + \theta_H h_1 + \theta_S s_1 + \theta_P p_1 + \theta_T t_3)z_1 + \\
&\ldots \text{(the above block repeated for } p_2\text{)} \\
&(\theta_Z z_1 + \theta_H h_1 + \theta_S s_2 + \theta_P p_3 + \theta_T t_4)z_1 + \\
&(\theta_Z z_1 + \theta_H h_1 + \theta_S s_2 + \theta_P p_3 + \theta_T t_5)z_1 + \\
&\ldots \text{(all above repeated for } h_2 \text{ and } h_3\text{)} \\
&(\theta_Z z_2 + \theta_H h_4 + \theta_S s_2 + \theta_P p_4 + \theta_T t_4)z_2 + \\
&(\theta_Z z_2 + \theta_H h_4 + \theta_S s_2 + \theta_P p_4 + \theta_T t_5)z_2.
\end{aligned}$$

The aggregate S_Z can be reformulated using the rewritings $\sum_{i=1}^{n} x \to x \cdot n$ and $\sum_{i=1}^{n} x \cdot a_i \to x \cdot \sum_{i=1}^{n} a_i$:

$$\begin{aligned}
S_Z = \theta_Z[&z_1(\underbrace{z_1 + \cdots + z_1}_{|\sigma_{Z=z_1}(TD)|}) + z_2(\underbrace{z_2 + \cdots + z_2}_{|\sigma_{Z=z_2}(TD)|})]+ \\
&\theta_H[z_1(\sum_{i=1}^{3} \underbrace{h_i + \cdots + h_i}_{|\sigma_{Z=z_1,H=h_i}(TD)|}) + z_2(\underbrace{h_4 + \cdots + h_4}_{|\sigma_{Z=z_2,H=h_4}(TD)|})]+ \\
&\theta_S[z_1(\sum_{i=1}^{2} \underbrace{s_i + \cdots + s_i}_{|\sigma_{Z=z_1,S=s_i}(TD)|}) + z_2(\underbrace{s_2 + \cdots + s_2}_{|\sigma_{Z=z_2,S=s_2}(TD)|})]+ \\
&\theta_P[z_1(\sum_{i=1}^{3} \underbrace{p_i + \cdots + p_i}_{|\sigma_{Z=z_1,P=p_i}(TD)|}) + z_2(\underbrace{p_4 + \cdots + p_4}_{|\sigma_{Z=z_2,P=p_4}(TD)|})]+ \\
&\theta_T[z_1(\sum_{i=1}^{5} \underbrace{t_i + \cdots + t_i}_{|\sigma_{Z=z_1,T=t_i}(TD)|}) + z_2(\sum_{i=4}^{5} \underbrace{t_i + \cdots + t_i}_{|\sigma_{Z=z_2,T=t_i}(TD)|})].
\end{aligned}$$

We then obtain the following cofactors in the sum S_Z:

$$\begin{aligned}
\text{Cofactor}[Z,Z] &= z_1 \cdot 24z_1 + z_2 \cdot 2z_2 \\
\text{Cofactor}[H,Z] &= z_1 \cdot 8(h_1 + h_2 + h_3) + z_2 \cdot 2h_4. \\
\text{Cofactor}[S,Z] &= z_1 \cdot 3(s_1 + s_2) + z_2 \cdot s_2. \\
\text{Cofactor}[P,Z] &= z_1 \cdot 3[3(p_1 + p_2) + 2p_3] + z_2 \cdot 2p_4. \\
\text{Cofactor}[T,Z] &= z_1 \cdot 3[2(t_1 + t_2 + t_3) + t_4 + t_5] + z_2 \cdot (t_4 + t_5). \;\square
\end{aligned}$$

This arithmetic factorisation is not arbitrary. It considers the arithmetic expressions grouped by the join Z-values, as done by the f-tree in Figure 2.1(b) for Z-singletons. Each cofactor in S_Z is expressed as a sum of terms with one term per each join Z-value z_1 and z_2. The numerical values occurring in the cofactors represent *occurrence counts*, e.g., 24 in $\psi_z = 24z_1$ states that z_1 occurs in 24 tuples in the training dataset, while 8 in $\psi_h = 8(h_1+h_2+h_3)$ states that each of h_1, h_2, and h_3 occurs in 8 tuples with z_1. The expressions ψ_z and ψ_h represent *sums* of Z-values and respectively H-values that occur in the same tuples with z_1 and are *weighted* by their occurrence counts.

The above rewritings are sufficient for cofactors of features θ_X in Sum_Y, where X and Y are from the same input relation. They do not, however, capture the full spectrum of possible computational saving. Moreover, they do not bring asymptotic savings. In case X and Y are from different input relations, then we can potentially save more computation.

Example 3.5 Consider now a rewriting of the cofactor of parameter θ_P in sum S_T:

$$\begin{aligned}
\text{Cofactor}[P,T] = &\ 3[(p_1 + p_2) \cdot (t_1 + t_2 + t_3)] + 3[p_3 \cdot (t_4 + t_5)]+ \\
&\ p_4 \cdot (t_4 + t_5).
\end{aligned}$$

The three terms in the outermost sum correspond to different pairs of join values for Z and S, namely (z_1, s_1), (z_1, s_2), and (z_2, s_2). This rewriting thus follows the same join order as the f-tree in Figure 2.1(b). These terms read as follows: Each of the P-values p_1 and p_2 occurs in three tuples with each of the T-values t_1 to t_3; the P-value p_3 occurs in three tuples with each of the T-values t_4 and t_5; the P-value p_4 occurs in one tuple with each of the T-values t_4 and t_5. $\square$

The rewritten expression for Cofactor$[P,T]$ factors out sums, e.g., $p_1 + p_2$, using a rewriting more powerful than those for the sum S_Z given in Example 3.4:

$$\sum_{i=1}^{r}\sum_{j=1}^{s}(x_i \cdot\ y_j) \rightarrow (\sum_{i=1}^{r} x_i) \cdot (\sum_{j=1}^{s} y_j)$$

This rewriting can transform expressions to exponentially smaller equivalent ones.

The above rewritings are already implemented by the factorised join from Figure 2.1(c). For instance, the sums of values in the cofactors mentioned in Examples 3.4 and 3.5 can be recovered via unions of their corresponding singletons in the factorisation. Since Z-singletons are above the singletons for the other attributes, they are in one-to-many relationships with singletons for the other attributes. This explains the rewritings in Example 3.4 for the cofactors in sum S_Z: Each of z_1 and z_2 are paired with the weighted sums of all H-values underneath, namely $8(h_1+h_2+h_3)$ and respectively $2h_4$. Similar pairings are with the weighted sums of values for each of the other attribute. Since the singletons for P and T are on different branches in the f-tree, a Cartesian product of a union of P-singletons and a union of T-singletons becomes a product of the sums of the corresponding P-values and T-values.

To compute the cofactors, we need to compute the occurrence counts and, based on them, the weighted sums used in expressing the cofactors. As we discuss in the next section, this can be done in one pass over the factorisation.

3.1 Rewriting the Parameter Cofactors

3.2 Cofactor Computation over any Factorisation

We compute the cofactors in two passes over the factorised join. Figure 3.1 gives an algorithm for each of the two computation passes. They recursively traverse the factorisation and collect counts, schemas, weighted sums, and partial cofactors for each node in the factorisation. In the sequel, for a factorisation E, we denote by $[\![E]\!]$ the relation it represents.

In the first pass, we build the F-layer on top of the factorisation. It records for each node, or equivalently for the factorisation E rooted at that node: (1) the number of tuples (Count) in the relation $[\![E]\!]$; (2) the schema (Schema) of the relation $[\![E]\!]$; and (3) for each attribute $A \in$ Schema(E), the sum (WSum) of all A-values in E weighted by their occurrence counts in E. The structural patterns matching the node are given on the leftmost column in Figure 3.1. There are two subtle points. All children of a union have the same schema, which is also the schema of the union. The Count values are used to compute occurrence counts as follows. If E is the child of a product $E_\times$, its occurrence count in $E_\times$ (i.e., the occurrence count of each tuple in $[\![E]\!]$) is the product of the Count values of its siblings. This is correct, since each of the tuples represented by E is extended in the relation $[\![E_\times]\!]$ by each tuple represented by each of E's siblings.

Pattern	*Algorithm*	
switch E:	**build-Flayer**(Factorization E)	**compute-cofactor**(Factorization E, int *multiplicity*)
$\langle A : a \rangle$	$\mathsf{Count}[E] = 1$ $\mathsf{Schema}[E] = \{A\}$ $\mathsf{WSum}[E, A] = a$	$\text{Cofactor}[A, A] \mathrel{+}= \textit{multiplicity} \cdot a^2$
$\langle A : a \rangle$ $\times$ E_1	**build-Flayer**(E_1) $\mathsf{Count}[E] = \mathsf{Count}[E_1]$ $\mathsf{Schema}[E] = \{A\} \cup \mathsf{Schema}[E_1]$ $\mathsf{WSum}[E, A] = a \cdot \mathsf{Count}[E_1]$ **for** $B \in \mathsf{Schema}[E_1]$ **do** $\mathsf{WSum}[E, B] = \mathsf{WSum}[E_1, B]$	**compute-cofactor**(E_1, *multiplicity*) $\text{Cofactor}[A, A] \mathrel{+}= \textit{multiplicity} \cdot a^2 \cdot \mathsf{Count}[E_1]$ **for** $B \in \mathsf{Schema}[E_1]$ **do** $\text{Cofactor}[A, B] \mathrel{+}= \textit{multiplicity} \cdot a \cdot \mathsf{WSum}[E_1, B]$

$\cup$
$/ \ \backslash$
$E_1 \cdots E_n$
	for $1 \leq i \leq n$ **do build-Flayer**(E_i)
	$\mathsf{Count}[E] = \sum_{i=1}^{n} \mathsf{Count}[E_i]$
	$\mathsf{Schema}[E] = \mathsf{Schema}[E_1]$
	for $1 \leq i \leq n$ **do**
	  for $B \in \mathsf{Schema}[E]$ **do**
	    $\mathsf{WSum}[E, B] \mathrel{+}= \mathsf{WSum}[E_i, B]$
		for $1 \leq i \leq n$ **do**
		  compute-cofactor(E_i, *multiplicity*)

$\times$
$/ \ \backslash$
$E_1 \cdots E_n$
	for $1 \leq i \leq n$ **do build-Flayer**(E_i)
	$\mathsf{Count}[E] = \Pi_{i=1}^{n} \mathsf{Count}[E_i]$
	$\mathsf{Schema}[E] = \bigcup_{i=1}^{n} \mathsf{Schema}[E_i]$
	for $1 \leq i \leq n$ **do**
	  $C_{\text{-}i} = \mathsf{Count}[E] / \mathsf{Count}[E_i]$
	  for $B \in \mathsf{Schema}[E_i]$ **do**
	    $\mathsf{WSum}[E, B] = C_{\text{-}i} \cdot \mathsf{WSum}[E_i, B]$
		$C = \textit{multiplicity} \cdot \mathsf{Count}[E]$
		for $1 \leq i \leq n$ **do**
		  $C_{\text{-}i} = C / \mathsf{Count}[E_i]$
		  compute-cofactor($E_i, C_{\text{-}i}$)
		  for $i < j \leq n$ **do**
		    $C_{\text{-}ij} = C_{\text{-}i} / \mathsf{Count}[E_j]$
		    for $A \in \mathsf{Schema}[E_i], B \in \mathsf{Schema}[E_j]$ **do**
		      $\text{Cofactor}[A, B] \mathrel{+}= C_{\text{-}ij} \cdot \mathsf{WSum}[E_i, A] \cdot \mathsf{WSum}[E_j, B]$

Figure 3.1: F-layer and cofactor computation for a factorisation E. Multiplicity represents the number of possible extensions of tuples in $[\![E]\!]$ to tuples in the entire training dataset. Both algorithms use top-down pattern matching on E: the patterns are in the left column and the corresponding actions are in the other two columns. The algorithms return after executing the action associated with a matched pattern.

Example 3.6 Figure 3.2 shows the F-layer for the factorisation in Figure 2.1(c). The numbers in circles are the counts and the expressions in rectangles are the weighted sums for each attribute at a factorisation node. The counts and weighted sums are computed bottom-up. □

In the second pass, we incrementally compute the cofactors for all parameters in all sums. Since the cofactor matrix is commutative (cf. Proposition 3.1), it is sufficient to only compute the cofactors for A and B, where A occurs before B in the depth-first left-to-right preorder traversal of the f-tree (and factorisation). Since it commutes with union, we compute partial cofactors for each descendant node of a union and then add them up. Initially, all cofactors are 0.

The occurrence count of a factorisation node consists of a component that depends on the factorisation fragments within the factorisation, as well as an component for factorisation fragments outside of it. As we descend down the factorisation, we maintain at each node ν the *multiplicity* of its represented tuples, which accounts for the component of the occurrence count due to factorisation fragments that are outside the current node ν. The second component of the occurrence count is due to factorisation fragments within ν and is already captured by the value of Count and the weighted sums at ν.

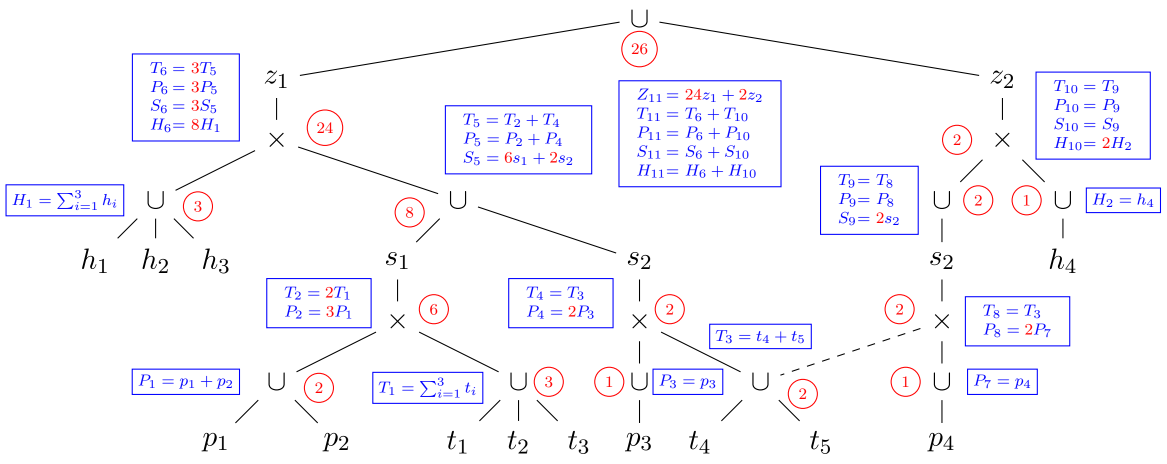

$T_6 = 3T_5$ $P_6 = 3P_5$ $S_6 = 3S_5$ $H_6 = 8H_1$
$T_5 = T_2 + T_4$ $P_5 = P_2 + P_4$ $S_5 = 6s_1 + 2s_2$
$Z_{11} = 24z_1 + 2z_2$ $T_{11} = T_6 + T_{10}$ $P_{11} = P_6 + P_{10}$ $S_{11} = S_6 + S_{10}$ $H_{11} = H_6 + H_{10}$
$T_{10} = T_9$ $P_{10} = P_9$ $S_{10} = S_9$ $H_{10} = 2H_2$
$T_9 = T_8$ $P_9 = P_8$ $S_9 = 2s_2$
$H_1 = \sum_{i=1}^{3} h_i$
$H_2 = h_4$
$T_2 = 2T_1$ $P_2 = 3P_1$
$T_4 = T_3$ $P_4 = 2P_3$
$T_3 = t_4 + t_5$
$T_8 = T_3$ $P_8 = 2P_7$
$P_1 = p_1 + p_2$
$T_1 = \sum_{i=1}^{3} t_i$
$P_3 = p_3$
$P_7 = p_4$
26
24
8
3
6
2
1
z_1
z_2
h_1 h_2 h_3 h_4
s_1 s_2
p_1 p_2 p_3 p_4
t_1 t_2 t_3 t_4 t_5

Figure 3.2: The F-layer for the factorised relation in Figure 2.1(c) as computed by the build-Flayer algorithm.

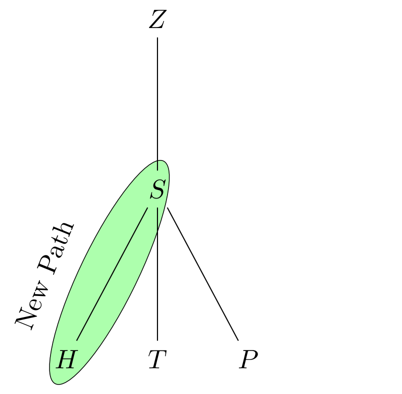
	θ_Z	θ_S	θ_P	θ_T	θ_H	θ_0
Σ_Z	$24z_1^2 + 2z_2^2$	$z_1S_6 + z_2S_{10}$	$z_1P_6 + z_2P_{10}$	$z_1T_6 + z_2T_{10}$	$z_1H_6 + z_2H_{10}$	$24z_1 + 2z_2$
Σ_S	Σ_Z/θ_S	$18s_1^2 + 6s_2^2 + 2s_2^2$	$3s_1P_2 + 3s_2P_4 + s_2P_8$	$3s_1T_2 + 3s_2T_4 + s_2T_8$	$S_5H_1 + S_9H_2$	$6s_1 + 2s_2 + 2s_2$
Σ_P	Σ_Z/θ_P	Σ_S/θ_P	$9p_1^2 + 9p_2^2 + 6p_3^2 + 3p_4^2$	$3P_1T_1 + 3P_3T_3 + P_7T_3$	$P_5H_1 + P_9H_2$	$p_1 + p_2 + p_3 + p_4$
Σ_T	Σ_Z/θ_T	Σ_S/θ_T	Σ_P/θ_T	$6t_1^2 + 6t_2^2 + 6t_3^2 + 4(t_4^2 + t_5^2)$	$T_5H_1 + T_9H_2$	$t_1 + t_2 + t_3 + 2(t_4 + t_5)$
Σ_H	Σ_Z/θ_H	Σ_S/θ_H	Σ_P/θ_H	Σ_T/θ_H	$8h_1^2 + 8h_2^2 + 8h_3^2 + 2h_4^2$	$h_1 + h_2 + h_3 + h_4$
Σ_0	Σ_Z/θ_0	Σ_S/θ_0	Σ_P/θ_0	Σ_T/θ_0	Σ_H/θ_0	26

Figure 3.3: Cofactor matrix based on the F-layer from our running example. Once computed, we may choose the set of features and the label to predict. The parameter for the label is then fixed to -1.

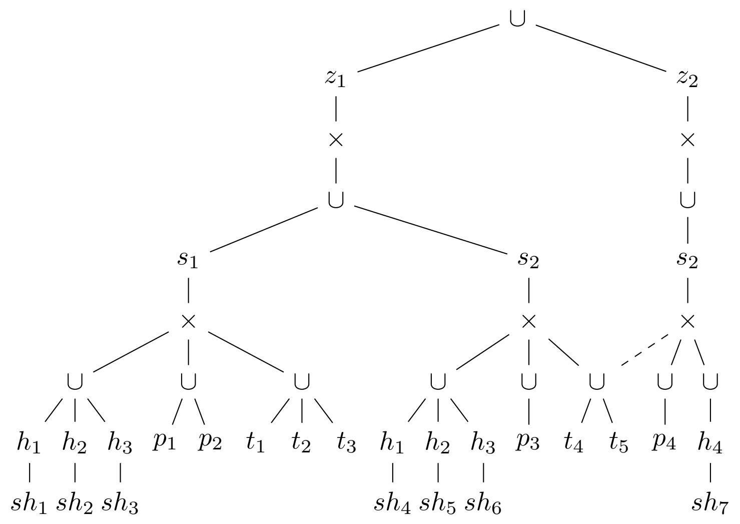
The update of cofactors at a product node ν is most challenging. For two attributes A and B from different branches E_i and respectively E_j (i.e., A and B are independent in the f-tree), we add to Cofactor$[A, B]$ the product of the weighted sums of A at node E_i and of B at node E_j, and their joint multiplicity, which is the product of the counts of the remaining siblings. The product of the weighted sums stands for all possible combinations c of A-values and B-values in the relation represented by ν, whereas their joint multiplicity stands for the number of possible combinations of c in the relation represented by the overall factorisation.

Example 3.7 We continue our example and use the F-layer in Figure 3.2 to compute the matrix of cofactors in Figure 3.3. We traverse the factorisation depth-first left-to-right. When we encounter the H-singleton leaves, we update Cofactor$[H, H]$; similarly for all leaves of an attribute. The product node under s_1 has the schema $\{P, T\}$ and updates Cofactor$[T, P]$ with the product of the weighted sums T_1 and P_1 for T and respectively P at that node, and of the joint multiplicity 3 of T_1 and P_1, which is the count of the other branch of the closest ancestor that is a product node. Further additions to this cofactor happen when we reach the product nodes under the two s_2 nodes. Cofactor$[T, Z]$ is updated twice: when reaching z_1 and z_2 since Z is above T in the factorisation, when we add the product of z_1 (z_2) and of the weighted sum of T under z_1 (respectively z_2). The multiplicities are one in both cases. $\square$

If the regression problems also learns an intercept, then it can be represented by a virtual singleton $\langle I : 1\rangle$ on top of the factorisation and by a virtual node I as root in the f-tree. This would be equivalent to adding a column of ones to the design



(a) F-tree F with path constraint.



∪
z_1
z_2
×
∪
s_1
s_2
s_2
h_1 h_2 h_3 p_1 p_2 t_1 t_2 t_3 h_1 h_2 h_3 p_3 t_4 t_5 p_4 h_4
sh_1 sh_2 sh_3 sh_4 sh_5 sh_6 sh_7

(b) Factorisation $F(\mathbf{D})$ of $Q_\sigma(\mathbf{D})$ over F.

Figure 3.4: Factorised database D from Figure 2.1(a) which enforces the path constraint to include the interaction term $\phi_k(s_i, h_j) = s_i \cdot h_j$, which are shown by singletons sh_i in the factorisation.

matrix, which is typically done to learn the intercept. Both the F-layer and cofactor computation would then work for the intercept as well. Figure 3.3 shows the cofactors for the intercept (denoted by θ_0 in the matrix). In particular, Cofactor$[0, 0]$ is the number of tuples in the relation represented by the entire factorisation.

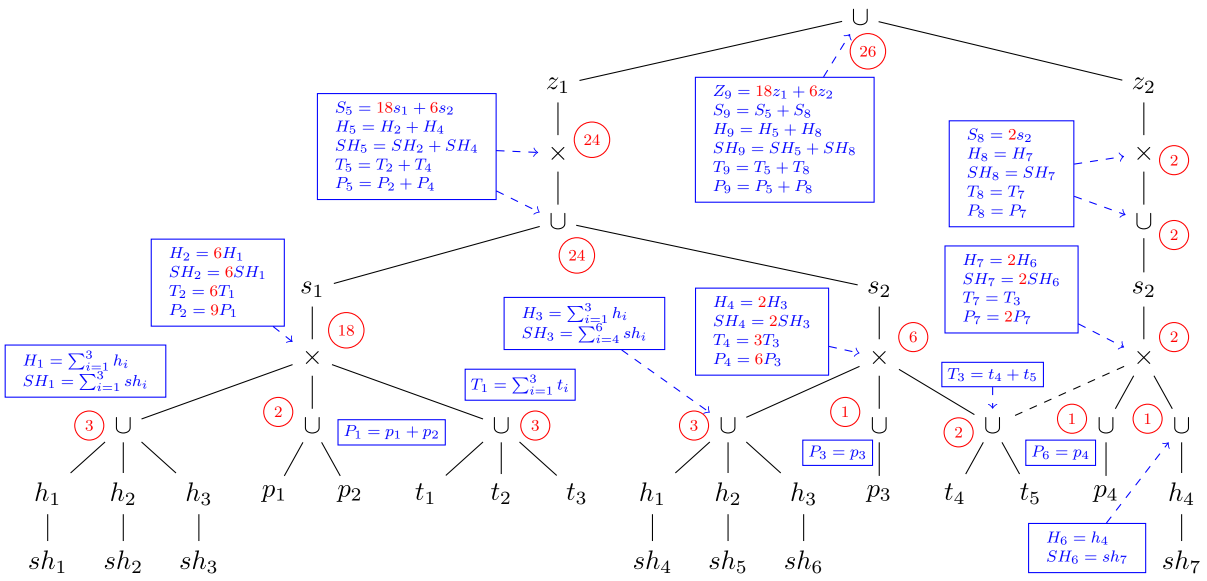
At each node ν, both algorithms recurse once for each child of ν and we need time linear in its schema to build the F-layer and at most quadratic in its schema to compute cofactors. For the latter, the amortised cost per pair (A, B) of attributes for which we compute the cofactor is however linear, since the more combinations of attributes we consider at a product node, the less remains to be considered at other nodes along the root-to-leaf path in the factorisation.

3.3 Extending F with non-linear basis functions

It is possible to extend the regression learner **F** with any of the non-linear basis functions that were discussed in Section 2.2. This implies that **F** does not only learn linear models, but can also arbitrary complex functions. The basis functions we have considered can be grouped in two categories: basis functions over a single feature, and basis functions over multiple features.

Basis functions over a single feature x_k can be supported trivially in any factorisation, because they are local modifications at the value level which are not dependent on other attributes. Polynomial terms with degree d, for example, can be included in the regression model by adding singletons $\langle \phi_k : x_k^d \rangle$ under each singleton x_k in the factorisation.

Basis functions over multiple features, such as feature interactions, on the other hand, are challenging as they may restrict the factorisation structure. For instance, the basis function $\phi_k(x_i, x_j) = x_i \cdot x_j$ can only be supported efficiently by f-trees where the attributes x_i and x_j are along the same root-to-leaf path as if they were attributes of a same relation, since we require to compute all possible combinations of values for



$Z_9 = 18z_1 + 6z_2$, $S_9 = S_5 + S_8$, $H_9 = H_5 + H_8$, $SH_9 = SH_5 + SH_8$, $T_9 = T_5 + T_8$, $P_9 = P_5 + P_8$
$S_5 = 18s_1 + 6s_2$, $H_5 = H_2 + H_4$, $SH_5 = SH_2 + SH_4$, $T_5 = T_2 + T_4$, $P_5 = P_2 + P_4$
$S_8 = 2s_2$, $H_8 = H_7$, $SH_8 = SH_7$, $T_8 = T_7$, $P_8 = P_7$
$H_7 = 2H_6$, $SH_7 = 2SH_6$, $T_7 = T_3$, $P_7 = 2P_7$
$H_2 = 6H_1$, $SH_2 = 6SH_1$, $T_2 = 6T_1$, $P_2 = 9P_1$
$H_4 = 2H_3$, $SH_4 = 2SH_3$, $T_4 = 3T_3$, $P_4 = 6P_3$
$H_1 = \sum_{i=1}^3 h_i$, $SH_1 = \sum_{i=1}^3 sh_i$
$H_3 = \sum_{i=1}^3 h_i$, $SH_3 = \sum_{i=4}^6 sh_i$
$T_1 = \sum_{i=1}^3 t_i$
$T_3 = t_4 + t_5$
$P_1 = p_1 + p_2$
$P_3 = p_3$
$P_6 = p_4$
$H_6 = h_4$, $SH_6 = sh_7$

Figure 3.5: F-layer for Factorisation $F(\mathbf{D})$ from Figure 3.4(b).

	θ_Z	θ_S	θ_P	θ_T	θ_H	θ_{SH}	θ_0
Σ_Z	$24z_1^2 + 2z_2^2$	$z_1S_5 + z_2S_8$	$z_1P_5 + z_2P_8$	$z_1T_5 + z_2T_8$	$z_1H_5 + z_2H_8$	$z_1SH_5 + z_2SH_8$	$24z_1 + 2z_2$
Σ_S	Σ_Z/θ_S	$18s_1^2 + 6s_2^2 + 2s_2^2$	$s_1P_2 + s_2P_4 +$ s_2P_7	$s_1T_2 + s_2T_4 +$ s_2T_7	$s_1H_2 + s_2h_4 + s_2h_7$	$s_1SH_2 + s_2SH_4 + s_2SH_7$	$18s_1 + 6s_2 + 6s_2$
Σ_P	Σ_Z/θ_P	Σ_S/θ_P	$9p_1^2 + 9p_2^2 +$ $6p_3^2 + 2p_4^2$	$3P_1T_1 + 3P_3T_3 +$ P_6T_3	$3P_1H_1 + 2P_3H_3 + P_6H_6$	$3P_1SH_1 + 2P_3SH_3 + P_6SH_6$	$9p_1 + 9p_2 + 6p_3 + 2p_4$
Σ_T	Σ_Z/θ_T	Σ_S/θ_T	Σ_P/θ_T	$6(t_1^2 + t_2^2 + t_3^2) +$ $4(t_4^2 + t_5^2)$	$2T_1H_1 + T_3H_3 + T_3H_6$	$2T_1SH_1 + T_3SH_3 + T_3SH_6$	$3(t_1 + t_2 + t_3)$ $+4(t_4 + t_5)$
Σ_H	Σ_Z/θ_H	Σ_S/θ_H	Σ_P/θ_H	Σ_T/θ_H	$6(h_1^2 + h_2^2 + h_3^2) +$ $2(h_1^2 + h_2^2 + h_3^2) + 2h_4^2$	$6(h_1sh_1 + h_2sh_2 + h_3sh_3) +$ $2(h_1sh_4 + h_2sh_5 + h_3sh_6) + 2h_4sh_7$	$6(h_1 + h_2 + h_3) +$ $2(h_1 + h_2 + h_3) + 2h_4$
Σ_{SH}	Σ_Z/θ_{SH}	Σ_S/θ_{SH}	Σ_P/θ_{SH}	Σ_T/θ_{SH}	Σ_H/θ_{SH}	$6(sh_1^2 + sh_2^2 + sh_3^2) +$ $2(sh_4^2 + sh_5^2 + sh_6^2) + 2sh_7^2$	$6(sh_1 + sh_2 + sh_3) +$ $2(sh_4 + sh_5 + sh_6) + 2sh_7$
Σ_0	Σ_Z/θ_0	Σ_S/θ_0	Σ_P/θ_0	Σ_T/θ_0	Σ_H/θ_0	Σ_{SH}/θ_0	26

Figure 3.6: Cofactor matrix based on the F-layer from Figure 3.5[1].

x_i and x_j. We can enforce this path constraint by enriching the database with one (not materialised) relation over the schema $R_k(x_i, x_j)$ and the query Q with a natural join with R_k. The f-trees for the enriched query will necessarily satisfy the new path constraint and the factorisation will have singletons for every combination of values for x_i and x_j along a same path. We can thus add to the factorisation a singleton $\langle \phi_k : \phi_k(x_i, x_j) \rangle$ under each pair of singletons for x_i and x_j.

Theorem 3.3 can be rephrased for linear functions with basis functions as follows. We say that the basis functions $\phi_0, \ldots, \phi_b$ over the sets of features $S_0, \ldots, S_b$ *induce* a relational schema $\sigma = (R_0(S_0), \ldots, R_b(S_b))$. Given a join query Q and the above schema σ, an *extension of* Q *with respect to* σ is a join query $Q_\sigma = Q \bowtie R_0 \bowtie \cdots \bowtie R_b$.

Theorem 3.8 *Let Q be a join query and $\mathbf{D}$ a database that define the training dataset $Q(\mathbf{D})$, and f_θ a linear function with basis functions that induce a relational schema σ. Let Q_σ be the extension of Q with respect to σ. Then, the parameters of f_θ can be learned in time $O(|\mathbf{D}|^{fhtw(Q_\sigma)})$.*

The proof of Theorem 3.8 is provided in Appendix. By means of an example, we illustrate how feature interactions can restrict the factorisation and how models with non-linear basis functions are learned in $\mathbf{F}$.

[1]For better readability this figure has also been added to the Appendix.

Example 3.9 In Section 3.2, we use the training dataset presented in Figure 2.1(a) to show how factorised joins and an algebraic rewriting of the objective function can be exploited to learn a linear least squares regression model with only two passes over the factorisation. We will now show how this example can be extended to learn the parameters for interaction terms as well.

Assuming that we would like to understand the relationship between attributes S and H, then we would add the interaction term $\phi_k(s_i, h_j) = s_i \cdot h_j$ to our feature space. Since S and H are on different paths in the f-tree, it is necessary to add the virtual relation $R_k(s, h)$ to enforce a path constraint between the two attributes. This will ensure that attributes S and H are along the same path in the f-tree as well as the factorisation. Figure 3.4 shows the modified f-tree, and the resulting factorised join, that accounts for this restriction. The factorisation also includes the singletons sh_i, which are the result of adding the interaction term to the feature space. Figure 3.5 and Figure 3.6 show the computation of the F-layer and the cofactor matrix for this new factorisation. □

3.4 F*: Pushing the F-layer into Factorised Join Computation

In previous sections, it has been shown that it is possible to learn least squares regression models for any factorisation by traversing the factorisation twice (once to create the F-layer and once to compute the cofactor matrix). If the input data, however, is not given as a factorisation but rather as relational tables, then it is possible to merge the computation of the factorised join and the F-layer. Consequently, instead of creating the F-layer over the factorised join, the factorisation itself becomes the F-layer. This is the underlying framework of **F***, an optimised version of **F**. **F*** is a stand-alone, fully integrated end-to-end system that takes any number of relations as input and outputs a learned parameterised regression model.

Since the computation of the F-layer can be pushed into the computation of the join, the regression learner only requires one more pass over the resulting factorisation to compute the cofactor matrix. As the computation for the cofactor matrix remains the same as described in Section 3.2, we only show how the computation of the F-layer can be pushed into the factorised join algorithm.

3.4.1 Computing the F-layer in F*

The algorithm that creates the factorisation for this optimised approach is based on a worst-case optimal multiway sort merge join, similar to the one used in the query engine FDB [6]. This join algorithm is then modified to incorporate the computations for the F-layer, which have been outlined in Figure 3.1. The algorithm used to compute the F-layer in **F** has four components, which are symbolised as the four cases in the algorithm that match the switch statement. In this section we show how each of these four cases can be incorporated in the factorised join algorithm.

The key insight for our approach is that the multiway sort merge join algorithm implemented in FDB recursively creates the nodes in the factorisation. This means that the creation of the factorisation resembles the recursive structure that was used to construct the F-layer for **F**.

```
Input: Database $D$ with relations $R$, FTree $T$, Query $Q$
Sort all $R$ according to join order
$lowPtr[\,]$, $upPtr[\,] \leftarrow$ pointers to first, last tuple of each relation
Call buildExpression(F$\rightarrow$root, $lowPtr$, $upPtr$)
function: buildExpression(FTreeNode $node$, lowerBound, upperBound)
  $A \leftarrow$ attribute of $node$
  for Relations $R$ with Attribute $A$ do
    lower[$R$] $\leftarrow$ lowerBound[$R$]
    upper[$R$] $\leftarrow$ highest tuple in $R$ with same value for $A$ as lower[$R$]
  $E_U \leftarrow$ empty summation expression
  while $\nexists R \in D$ lower[$R$] = upperBound[$R$] do
    $a \leftarrow$ value for $A$ in lower[$R$] in relation $R \in D : \nexists R' \in D :$ lower[$R'$] < lower[$R$]
    if $\forall\, R \in D$, lower[$R$] $= a$
      $S \leftarrow$ singleton $\langle A : a \rangle$
      if $node$ has children $\{c_1 \dots c_n\}$
        for $1 \leq i \leq n$ do $E_i$ = buildExpression($c_i$, lower, upper)
        if $\exists\, 1 \leq i \leq n$ s.t. $E_i = \emptyset$ : delete($E_1, \dots, E_n$) and continue with join
        $E_X \leftarrow E_1 \times E_2 \times \cdots \times E_n$
        $\mathsf{Count}[E_X] = \Pi_{i=1}^{n} \mathsf{Count}[E_i]$
        $\mathsf{Schema}[E_X] = \bigcup_{i=1}^{n} \mathsf{Schema}[E_i]$
        for $1 \leq i \leq n$ do
          $C_{\text{-}i} = \mathsf{Count}[E_X] / \mathsf{Count}[E_i]$
          for $B \in \mathsf{Schema}[E_i]$ do
            $\mathsf{WSum}[E_X, B] = C_{\text{-}i} \cdot \mathsf{WSum}[E_i, B]$
        $S \leftarrow \langle A : a \rangle \times E_X$
        $\mathsf{Count}[S] = \mathsf{Count}[E_X]$
        $\mathsf{Schema}[S] = \{A\} \cup \mathsf{Schema}[E_X]$
        $\mathsf{WSum}[S, A] = a \cdot \mathsf{Count}[E_X]$
        for $B \in \mathsf{Schema}[E_X]$ do $\mathsf{WSum}[S, B] = \mathsf{WSum}[E_X, B]$
      else
        $\mathsf{Count}[S] = 1$
        $\mathsf{Schema}[S] = \{A\}$
        $\mathsf{WSum}[S, A] = a$
      $E_U := E_U \times S$
      $\mathsf{Count}[E_U]$ += $\mathsf{Count}[S]$
      $\mathsf{Schema}[E_U] = \mathsf{Schema}[S]$
      for $B \in \mathsf{Schema}[S]$
        $\mathsf{WSum}[E_U, B]$ += $\mathsf{WSum}[S, B]$
    for Relations $R$ with Attribute $A$ and lower[$R$] $= a$
      lower[$R$] $\leftarrow$ tuple in $R$ after upper[$R$]
      upper[$R$] $\leftarrow$ highest tuple in $R$ with same value for $A$ as lower[$R$]
  if $E_U$ has no child : delete($E_U$) and return $\emptyset$
  else return $E_U$
```

Figure 3.7: Compute F-layer as part of Factorised Join Computation

We give a high level overview of the algorithm that merges the F-layer and join construction; the complete algorithm is presented in Figure 3.7. The multiway sort merge join component of the algorithm relies on the f-tree to define the nesting structure for the join. Each node in the f-tree defines an attribute A and we define $\langle A : a_i \rangle$ to be all singleton values that satisfy the join condition for all relations with attribute A.

The f-tree is traversed top down and at each node we create a union factorisation of the form $E_U = E_1 \cup \ldots \cup E_n$, where n is the number of singletons that satisfy the join condition. The structure of $E_1, \ldots, E_n$ depends on the whether the given f-tree node is a leaf or not. If it is a leaf, then there are no further factorisations to be added, which means that each $E_i = \langle A : a_i \rangle$. Otherwise, each E_i has the following structure: $E_i = \langle A : a_i \rangle \times E_X$, where $E_X = E'_1 \times \ldots \times E'_m$ is a product factorisation for which m is the number of children of the f-tree node and each E'_i is created by a recursive call for the next attribute in the f-tree.

This join algorithm lends itself well for the F-layer construction, because it allows for a seamless introduction of the four components of the algorithms from Figure 3.1 that is used for the F-layer construction. We present the connections between the components of the original F-layer algorithm and the optimised algorithm in the order in which the factorisation nodes are created in the join algorithm.

The union factorisation E_U is originally empty and then it is extended one-by-one by a factorisation E_i for each singleton $\langle A : a_i \rangle$. This means that the Count and WSum of E_U are updated each time a factorisation E_i is added to E_U by adding the values for Count and WSum of E_i to the corresponding attributes in E_U. The Schema is the same for all E_i, so that we can set the Schema for E_U equal to the schema of E_1. Once all the singletons are created, the algorithm has the same result as the algorithm shown in case three of the F-layer construction algorithm in Figure 3.1.

The update of the F-layer attributes for the singleton nodes $\langle A : a_i \rangle$ depends on the structure of the factorisation E_i. If E_i is of form $E_i = \langle A : a_i \rangle$, then we update the attributes as shown in case one of the F-layer algorithm; setting Schema $= \{A\}$, Count $= 1$ and WSum $= a$. If $E_i = \langle A : a_i \rangle \times E_X$, then the attributes are updated as specified in case two of the algorithm. This copies the value for Count and WSum from E_X and adds A to Schema and WSum.

The F-layer attributes of the product factorisation E_X are updated as shown in case four in the F-layer construction algorithm. The Count and Schema are set to the product, respective union, of the Count and Schema for all E'_i. Furthermore, the WSum is set to the corresponding WSum in E'_i for each attribute in the schema of E_X.

This high level description of the algorithm that computes the F-layer as part of the factorised join computation shows that the computations are identical to the F-layer computations outlined in Figure 3.1. For this reason, **F*** follows the same procedure as **F** and, therefore, also achieves the same result.

3.4.2 Strengths and Weaknesses of F*

F* is able to push the computation of the F-layer into the factorised join computation. This leads to a performance improvement in comparison to **F** because once the factorisation is computed the learning process only requires one more pass over the factorisation to compute the cofactor matrix. The creation of the F-layer on top of the factorisation, which requires a lot of object creations, has been mitigated.

Furthermore, **F*** has the advantage that the factorisation is solely used for learning least squares regression models, which means that the structure of the factorisation can be optimised to suit this one task. **F**, on the other hand, is integrated in the query engine FDB [6], which means that the factorisation that is used by **F** is used for a variety of query processing tasks. By scaling down the factorisation and ensuring that it is optimised for learning, the performance gain of **F*** can be improved further. In fact, it can be possible that the entire end-to-end solution is faster than the computation for the factorised join that is the input to **F**.

Although **F*** can be more performance efficient in comparison to **F**, it also restricts the flexibility of the end-to-end regression learner. Whereas **F** can work on top of any factorised database regardless how it is computed, **F*** only works for the specific setting where learning happens on top of a join of relational data. This lack of generality for **F*** is the price to pay for its improved performance over **F**.

Another limitation is that **F*** does not allow for an modification of the join output. For learning regression models, however, it can be beneficial to perform aggregates or other computations on the join to attain better features. This is explained by means of an example.

Example 3.10 In retail, it is often desirable to predict the next month's sales for a particular region or a collection of stores. Possible features for this model could be the sales data from previous months, to account for temporal trends, or also the number of promotions that stores offer in this region during the specified timeframe.

If the data is stored in a relational setting, however, then the raw data is not necessarily suitable to act as a feature for a model. For instance, sales data is commonly collected each day and for each store individually. In order to attain the monthly sales data for a region, it is necessary to perform aggregates over this data to sum over all data entries in the specific timeframe and location.

This is further complicated if aggregates need to be performed over attributes that are in different relations. Consider the second feature we mentioned: the number of promotions that the stores in the region offer during the specific timeframe under consideration. It is likely that the data is stored in a relation where each promotion is listed for a given store, which is identified by a unique ID. Therefore, in order to attain the data that is needed for this feature, it is necessary to perform an aggregate over the data to sum up all the promotions that fit the requirements. It is unlikely, however, that the store ID gives sufficient information about the location of the store. Therefore, the promotion relation would first need to be joined with another relation that would provide the information on where the store is located, so that further aggregates can be applied.

This exemplary scenario can be supported in FDB and consequently **F**, but it can not be done in **F***, which presents the inherent limitations of this proposed end-to-end system. □

Chapter 4

Implementation

Chapter 3 outlines the general framework of our regression learner **F** and its optimised version **F***. This chapter presents the details of the implementation of the algorithms, including some optimisations were used to lower the computational cost. In section 4.3, we present the details for the optimisation algorithm and our implementation of AdaGrad. The code is provided on the attached CD along with instructions on how to run it and a sample dataset for an exemplary learning problem. We also have a private GitHub repository for the team members. If you would like to access the repository, please contact one of the team members.

4.1 Implementation details for F

Our system **F** is a factorisation-aware regression learner that is implemented in C++ and fully integrated in the query engine FDB [6]. The input to the system is a factorised representation of the design matrix, which can be created by joining input relations with the worst-case optimal multiway sort merge join algorithm that comes with FDB.

The code for **F** is in provided in the `F` folder. The relevant classes can be found under the directory `src/ml`. The base class that implements the functionalities of **F** is the `LinearRegression` class. This class encapsulates the `Flayer` and `FlayerNode` classes, which implement the F-layer on top of the input factorisation.

One difference between the training example in Figure 2.1(c) and the concrete implementation is that there are no explicit singleton nodes in the implementation. The values for each attribute are stored inside the children nodes of a union. As a result, the factorisation only consists of unit and product nodes, as well as simple leaf nodes that store the singleton values when there are no further nodes to be added. As a result, The cases two and three from the algorithms that compute the F-layer and cofactor matrix in Figure 3.1 are merged into one single case. This merged case would be called whenever the root of the factorisation E is a union node.

The F-layer is implemented as an additional layer on top of the factorisation, which implies that the F-layer has the same structure as the factorisation. Each F-layer node, which is defined in the `FlayerNode` class, has a pointer to the corresponding node in the factorisation and extends the factorisation node with the F-layer attributes: Count, WSum, and Schema. We store the Count as an integer, Schema as an integer vector and WSum as an array of doubles.

The F-layer itself is constructed in the `Flayer` class, which implements two recurrent algorithms: one to create the F-layer nodes and the other to fill the attribute of the nodes. Contrary to the algorithm presented in Figure 3.1, we decided to separate the construction and the populating of the nodes, because the addition of basis functions can lead to modifications in the F-layer after it has been constructed. In order to decrease the memory usage as much as possible, the F-layer does not create nodes that correspond to leaf nodes in the factorisation, because they are not required for learning. All the necessary information for the algorithms can be stored in the directly preceding union node or retrieved through the pointer to the corresponding factorisation node.

The `LinearRegression` class encapsulates all the functionalities required to learn the parameters. The constructer is given an input factorisation and a text file that defines the features and label, including basis functions and interaction terms, that the model is based on. The constructor has four main components. First, it calls the `Flayer` class to create the F-layer based on the factorisation. Then it adds any additional nodes to the flayer, if there are interaction nodes in the model. Then it will call the `Flayer` class again to populate the attributes of the F-layer. Finally, cofactors are computed and stored in the cofactor matrix. We are able to populate the cofactors for the intercept on the fly so that no further modifications of the F-layer are required. Once the cofactor matrix is filled, the optimisation procedure (Section 4.3) proceeds to learn the parameters of the model.

4.1.1 Implementation of interaction terms in F

In order to add interaction nodes for features X_1 and X_2 to the model, it is important to ensure that the features are along the same path in the factorisation. This can be done by defining the f-tree manually in FDB so that it incorporates the required path constraint. The factorisation will then be created based on the nesting structure defined by the f-tree, including the path constraint required for the interaction term. Without loss of generality, we can assume that X_1 is above X_2 in the path where the interaction singletons should be added.

When the path constraints are provided, we add the interaction singletons $X_1 \cdot X_2$ locally below the singletons of feature X_2. This can be done by using the nesting structure of the f-tree to find the union factorisations for X_1. Then for every singleton node $\langle X_1 : x_1 \rangle$, we again use the f-tree to find the union factorisations for feature X_2 and directly underneath each singleton $\langle X_2 : x_2 \rangle$, we add the singletons $x_1 \cdot x_2$ to account for the interaction term.

This means that we do not have to traverse the entire factorisation but rather add the interaction terms locally. Once the interaction terms have been added in the factorisation, we can proceed to populate the F-layer attributes Count, Schema, and WSum.

4.2 Implementation details for F*

F* modifies the join algorithm from FDB, which implies that the system is not directly integrated in the query engine. Instead, it can be considered as a stand alone regression learner.

The code for **F*** is provided in the `f-star` folder on the CD. The base class for this implementation is the `LinearRegression` class in the `src/ml` folder. This class is built on the `FtreeBuilder` and the `FRepOverFtreeBuilder` classes, which respectively create the the f-tree and factorisation that also contains the F-layer.

The underlying algorithm that merges the construction of the F-layer and the join is presented in Figure 3.7. We outline a few subtle differences in the concrete implementation of **F*** to show where it differentiates from the implementations of FDB and **F**.

Similar to the F-layer in **F** it is not necessary to create a node for each leaf singleton value. Therefore, the join algorithm has been modified to create only one node at the leaf level to store all the information required for learning. This shows that factorisation used for learning can be scaled down to better suit the functionalities for the regression learner and to provide performance improvements.

Another difference in comparison to **F** is that **F*** directly populates the attributes of the F-layer during the construction of the factorisation. It is not clear, however, how additional nodes, such as the ones computed for interaction terms, can be pushed into the factorised join algorithm. Therefore, we add these additional nodes after the F-layer has been constructed and to modify the F-layer attributes (in particular Schema and WSum) of the preceding nodes in the factorisation accordingly. The Count attribute will not be affected, because there is only one additional node for this new feature added to each branch. This means that the addition of the extra node does not increase the number of tuples in the given factorisation.

The present implementation should be considered as a proof of concept. The implementation shows that it is in fact possible to push the F-layer into the join algorithm. Due to time constraints, however, the current system does not optimise the factorisation specifically for learning tasks. The general structure of the factorisation is still designed to suit a variety of query processing tasks, which is not necessarily optimal for learning regression models. For instance, children in a union node are currently implemented as linked lists, which is beneficial for query processing, but for learning it may be better to store the children in an array to allow for direct accessing. Future work will investigate how the structure of this factorisation can be modified to achieve the best possible performance for regression learning.

4.3 Implementation of learning and convergence

Our regression learner **F**, and its optimised version **F***, use the modified batch gradient descent algorithm from Chapter 3 to optimise the parameters of the model. Contrary to the standard batch gradient descent algorithm, our implementation does not traverse the input data during the optimisation procedure, but instead learns solely based on the cofactor matrix. Each row in the cofactor matrix gives the cofactors for the sum aggregate of one parameter, which means that the gradient of the objective function is given by the sum of the product of the parameters and the corresponding cofactors.

As mentioned in Section 2.2, the learning rate α in gradient descent optimisation can heavily influence the performance of the algorithm. The learning rate determines the rate of convergence and, if set correctly, ensures that the algorithm converges to the solution. If the learning rate is too large, however, it is possible that the algorithm will oscillate around the solution, but never reach it. We decided to use an adaptive learning rate, which dynamically updates the step size for each iteration, because this reduces the risk of setting a bad learning rate. The procedure we chose is AdaGrad [19], which is well-known and commonly used for a variety of applications.

AdaGrad dynamically updates the learning rate based on all previous iterations. This is done by dividing the initial learning rate by the sum of squared gradients of all previous iterations:

$$\alpha_{j,t} \;=\; \frac{\alpha}{\sqrt{\sum_{t'=1}^{t} g_{j,t'}^2}}$$

where $g_{t',j}$ is the gradient for parameter j at iteration t'.

The advantage of AdaGrad is that it allocates a different learning to each parameter, which makes the convergence procedure much more flexible. It also implies that the optimisation procedure can pay particular attention on infrequent, but highly predictive, features, without putting too much weight on frequent but less informative features. These properties make AdaGrad a well rounded procedure for choosing the learning rate. It mitigates the problem of predefining a fixed rate each time the algorithm is run on a different learning problem, and makes the optimisation procedure a lot more robust.

The only hyper-parameter for AdaGrad is α, which defines the step size at the very first iteration. This parameter has less significance than when the learning rate is fixed for all iterations, and, therefore, we can safely choose a large step size for the first iteration. In our implementation, α is set to 1.

Chapter 5

Experiments

In this section we report on the performance of an end-to-end solution for learning regression models over joins, which includes: (a) constructing the dataset via joins; (b) importing the dataset in the learning module; and (c) learning the parameters of regression functions. We show experimentally that the intermediate step of join computation is unnecessarily expensive. It entails a high degree of redundancy in both computation and data representation, yet this is not required for the end-to-end solution, whose result is a constant number of real-valued parameters. By factorising the join we reduce data redundancy while improving performance for both the join and the learning steps. Furthermore, a tight integration of the learning and the join processing modules eliminates the need for the data import step.

We present an empirical comparison of the proposed regression learner **F** and two two open-source state-of-the-art statistical systems: **P** (Python StatsModels [49]) and **R** [39]. These systems are comparable because they all require the materialisation of the join before running analytics. Contrary to **F**, **R** and **P** do not use an iterative approach to solve linear regression tasks, which shows that calculating the closed-form solution is commonly the optimisation method of choice. For **R** we used the *lm* (linear model) function which calculates the solution based on the QR-decomposition [20] (cf. Section 2.2). For **P** we used the *ols* (ordinary least squares) function, which calculates the ordinary least squares solution using the Moore-Penrose pseudoinverse [37].

We show that for a variety of datasets and learning tasks, **F** outperforms **R** and **P** by up to three orders of magnitude, while maintaining the accuracy of the other systems; we verified that the results (i.e., the learned parameters) of the three systems coincide with high precision. This performance boost is due to the three orthogonal optimisations used by **F**: (i) its adaptation to factorised data; (ii) decoupling of the convergence of function parameters from the computation of their cofactors; and (iii) shared computation of all cofactors in two passes over the factorised data. We also provide insights into the relative performance of **F**'s components: building the F-layer, cofactor computation, and convergence, with the first component taking the lion's share of the time due to object creation.

F* mitigates the object creation and, therefore, presents opportunities to improve the performance at learning time. An empirical comparison between **F** and **F*** shows the potential performance improvement by pushing the F-layer over the join.

We report wall-clock times representing the average of five runs for each of the three steps of the end-to-end learning solution, i.e., dataset construction by join materialisation, dataset import, and learning. For dataset construction, **F** uses the query engine FDB [6] to compute factorised joins. The input for both **R** and **P** is a flat relation computed by RDB, which is a lightweight main-memory multi-way sort-merge join algorithm that has been previously reported [6, 5] to outperform SQLite and PostgreSQL for the main-memory setting considered here. RDB comes in the same package with FDB. **F** has no import requirement as it is tightly integrated with FDB and works directly on the factorised join computed by FDB. **P** and **R** need to load the flat join result into their memory space, which requires one pass over the data and construction of an internal representation for the dataset.

In addition to the performance for learning, we report the performance for joins for reference only and to support the claim that the end-to-end performance of learning over joins is superior when considering succinct factorised joins. It is not the goal of this dissertation to propose novel join algorithms.

5.1 Experimental Setup

All experiments were performed on an Intel(R) Core(TM) i7-4770 8core/40GHz/64bit/32GB with Linux 3.13.0/g++4.8.4 (no compiler optimisation flags were used). All engines were run on one core and `ulimit` was set to unlimited.

5.1.1 Datasets and Learning Tasks

We experimented with a real-world dataset, which is used by a large US retailer for forecasting user demands and sales, and with two public datasets LastFM [12] and MovieLens [22], cf. Table 5.1. We also used a synthetic dataset modelling the textbook example on house price market [33]. We also considered learning over the Delicious [12] and Financial [7] public datasets, though we do not report them here since they bring no new insights.

The learning task requires to prepare the datasets. Firstly, we only kept features that represent quantities or Boolean flags over which we can learn and discarded string features (except if necessary for joins). Secondly, we normalised all number values of a feature A by mapping them to the $[0, 1]$ range of reals as follows. Let $\min_A$ and $\max_A$ be the minimum and respectively maximum value in the active domain of A. Then, a value v for A is normalised to $(v - \min_A)/\max_A$. Normalisation is essential so that all features have the same relative weight, e.g., avoiding that large date values represented in seconds since Jan 1, 1970 are more important than, say, small integer values representing the number of house bedrooms. It also preserves the cardinality of the join results from the original datasets.

To build the training dataset for the learning task, we considered in most cases the natural join of the normalised input tables since this brings together all relevant features. Since our primary goal is to benchmark the performance of the three systems, we instruct them to learn over all the present features. Experiment 8 discusses the performance for learning over a subset of the features.

We next briefly introduce the schemas and the learning tasks for all our datasets; they are detailed in the Appendix.

US Retailer

The *US retailer* dataset consists of three relations: `Inventory` (storing information about the inventory units for products in a location, at a given date; 84M tuples), `Census` (storing demographics information per zipcode such as population, median age, repartition per ethnicities, house units and how many are occupied, number of children per household, number of males, females, and families; 1293 tuples), and `Location` (storing for each zipcode distances to several other stores; 1317 tuples). The training dataset is the natural join of the three relations and has 31 features. We considered three linear regression tasks that predict the amount of inventory units based on all other features. The first one (L) considers the plain features, whereas the other two (N_1 and N_2) also consider interactions between features. N_1 considers two interactions of features from the same relation and for which no restructuring is necessary: (i) between median age and number of families (both from `Census`), and (ii) between different distances to other stores (both from `Location`). N_2 considers two interactions: (i) between population and number of house units (both from `Census`), and (ii) between median age and distance to another store (features of `Census` and `Location`, respectively). Thus, the f-tree used for N_2 needs to additionally satisfy the constraint that the features from its second interaction (population and house units) are on the same root-to-leaf path. For both N_1 and N_2, each new interaction term can be seen as a newly-derived feature for learning in addition to the initial 31 features, hence for each task we have $31 + 2 = 33$ features (cf. Table 5.1).

LastFM

LastFM [12] has three relations: `Userfriends` (pairing friends in the social network from the LastFM online music system; 25K tuples), `Userartists` (how often a user listens to a certain artist; 92K tuples), and `Usertaggedartiststimestamps` (the user classification of artists and the time when a user rated artists; 186K tuples). Our regression task is to predict how often a user would listen to an artist based on similar information for its friends. We consider two training datasets: L_1 joins two copies of `Userartists` with `Userfriends` to relate how often friends listen to the same artists; L_2 is L_1 where we also join in the `Usertaggedartiststimestamps` copies of both friends.

MovieLens

MovieLens [22] has three relations: `Users` (age, gender, occupation, zipcode of users; 6040 tuples), `Movies` (movie year and its type, e.g., action, adventure, animation, children, and so on; 3880 tuples), and `Ratings` (1M tuples) that users gave to movies on certain dates. The training dataset is the natural join of these tables and has 27 features. The regression task is to predict the rating given by a user to a movie.

Housing

Housing is a synthetic dataset emulating the textbook example for the house price market [33]. It consists of six tables: `House` (postcode, size of living room/kitchen area, price, number of bedrooms, bathrooms, garages and parking lots, etc.), `Shop` (postcode, opening hours, price range, brand, e.g., Costco, Tesco, Saynsbury's), `Institution` (postcode, type of educational institution, e.g., university or school, and number of students), `Restaurant` (postcode, opening hours, and price range), `Demographics` (postcode, average salary, rate of unemployment, criminality, and number of hospitals), and `Transport` (postcode, the number of bus lines, train stations, and distance to the city center for the postcode). The training dataset is the natural join of all relations (on postcode) and has 27 features. There are 25K postcodes, which appear in all relations. The scale factor s determines the number of generated distinct tuples per postcode in each relation: We generate s tuples in `House` and `Shop`, $\log_2 s$ tuples in `Institution`, $s/2$ in `Restaurant`, and one in each of `Demographics` and `Transport`. We considered three linear regression tasks that predict the price of a house based on all other features. The first one (L) considers the plain features, whereas the other two (N_1 and N_2) also consider interactions between features. N_1 considers two interactions: (i) between the type of house and the number of bedrooms (both from `House`), and (ii) between the number of train stations and the distance to city center (both from `Transport`). N_2 also considers two interactions: (i) between the number of bus lines and the shop opening hours (features of `Transport` and `Shop`, respectively), and (ii) between the size of educational institutions and the number of crimes per year (features of `Institutions` and `Demographics`, respectively). Thus, the f-tree used for N_2 needs to additionally satisfy an additional constraint per interaction.

5.2 Experimental Results

5.2.1 Flat vs. Factorised Joins: Compression Ratio

As shown in Table 5.1 and Figure 5.4(c), the compression factor brought by factorising the joins varies from 4.43 for MovieLens to 26.84 for the US retailer dataset, to over 10^2 for LastFM and over 10^3 for the synthetic dataset. For a scale factor s in the synthetic dataset, the natural join of all relations on postcode has $25\text{K} \times s^3/2 \times \log_2 s$ tuples, each of 27 singletons. In contrast to the flat join, the factorised join does not materialise the combinations (indeed, the Cartesian product) of houses, restaurants, institutions, and shops for each postcode, but instead keeps the data from each relation separately from the other relations. This makes the number of singletons of the factorised join bounded by the sum of the number of singletons in the input relations. The gap between the sizes of flat and factorised joins thus follows a quadratic function in the scale factor s, as confirmed by Figure 5.4(a) and Figure 5.4(c). To compute the number of singletons in the flat join beyond scale factor 16, we counted the number of tuples represented by the factorised join and multiplied it by the number of attributes, since RDB ran out of memory.

Feature interactions add additional constraints on the structure of the f-tree, which could lead to worse compression ratios. In our experiments, however, the effect of feature interactions on compression ratio was marginal. For US retailer, there is no change: for any location, the dataset records one distance to other shops and one median age, so a change in their order in the f-tree for task N_2 does not effect the number of singletons in the representation. For task N_2 on Housing, the interactions require a less optimal f-tree, which comes with a slight impact on the compression ratio: e.g., for scale factor 20 it degrades from 1.9K to 1.6K. Figure 5.3(b) shows that

		US retailer L	US retailer N_1	US retailer N_2	LastFM L_1	LastFM L_2	MovieLens L
# parameters		31	33	33	6	10	27
Join Size	**F** (via FDB)	96,328,820	96,328,820	96,328,820	2,577,556	2,332,036	6,092,286
	R&P (via RDB)	2,585,046,352	2,585,046,352	2,585,046,352	369,986,292	590,793,800	27,005,643
	Compression	26.84×	26.84×	26.84×	143.54×	253.34×	4.43×
Join Time	**F** (via FDB)	485.26	485.26	485.26	5.66	17.37	20.71
	R&P (via RDB)	3112.02*	3112.02*	3112.02*	313.65	611.80	43.17
	Speedup **R**/**F**	6.32×	6.32 ×	6.32×	55.41×	35.22×	2.08×
Import Time	**F**	0	0	0	0	0	0
	P	1164.40*	1164.40*	1164.40*	179.16	328.97	11.33
	R	1189.12*	1189.12*	1189.12*	155.91	276.77	10.86
Learn Time	**F**	9.69	9.82	9.79	0.53	0.89	3.87
	P	1199.50*	1277.10*	1275.08*	35.74	148.84	10.92
	R	810.66*	873.14*	884.47*	268.04	466.52	6.96
	Speedup **P**/**F**	**123.80**×	**130.06**×	**130.26**×	**67.18**×	**166.35**×	**2.82**×
	Speedup **R**/**F**	**83.67**×	**88.92**×	**90.36**×	**503.81**×	**521.41**×	**1.80**×
Total Time	**F**	494.95	495.08	495.05	6.19	18.26	24.58
	P	5475.92*	5553.52*	5551.50*	528.55	1089.61	65.42
	R	5111.80*	5174.28*	5185.61*	737.60	1355.09	60.99
	Speedup **P**/**F**	11.06×	11.22×	11.21×	85.36×	59.66×	2.66×
	Speedup **R**/**F**	10.32×	10.45×	10.47×	119.12×	74.19×	2.48×
F* Times	**F***	461.16	461.24	461.21	6.07	13.20	20.24
	Speedup **P**/**F***	11.87×	12.11×	12.03×	87.11×	82.56×	3.23×
	Speedup **R**/**F***	11.08×	11.28×	11.31×	121.57×	102.68×	3.01×

Table 5.1: Real datasets: Performance comparison for end-to-end learning solution (join, import, and building linear regression models) using **F**, **P**, and **R** (size in number of singletons, time in seconds). **P** and **R** crashed for US retailer due to memory limitation, the starred numbers are for running them on roughly equal-sized disjoint partitions of the location values for the join (four for **P** and ten for **R**) and adding up the times.

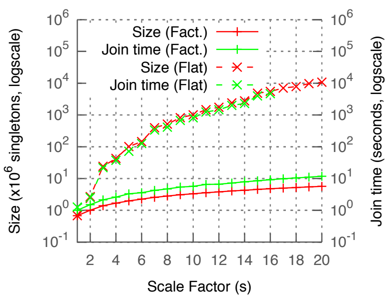
for scale factor 20 the interactions in N_2 can degrade the learning performance from about three to five seconds.

The compression ratio is a direct indicator of how well **F** fares against approaches that rely on flat representation of the training dataset, such as **P** and **R**, cf. Figure 5.4(b) for the synthetic dataset and Table 5.1 for the real-world datasets.

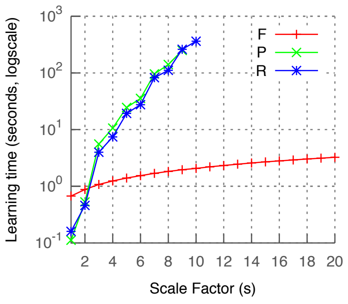
5.2.2 Flat vs. Factorised Joins: Performance

We verified that the speedup of the factorised join over the flat join follows the compression ratio, as reported in the literature [6]. For the synthetic dataset, this is depicted in Figures 5.4(a) and 5.4(c). After scale factor 16, we cannot compute anymore the flat join due to memory limitation (the subsequent learning with **R** and **P** already fails for scale 11). As shown in Figure 5.2, for scale 10 the factorised join is computed in six seconds vs. 800 seconds for the flat join.

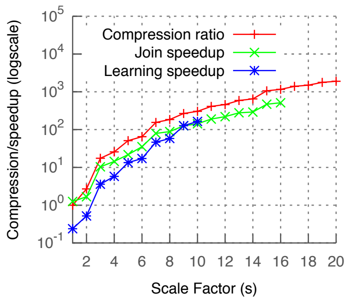
Table 5.1 reports the join time for the real-world datasets. The flat join of the US retailer dataset cannot be handled by our server; the relational engine got killed when trying to materialise the flat join to disk after the first 21GB tuples. In practice, users of such large datasets partition them and learn independently for each partition. This entails a loss of accuracy and misses correlations across features. Factorisation can effectively push the barrier of what is possible for large datasets orthogonally to approaches based on distribution. We partitioned the largest relation Inventory (84M tuples) into four (ten) disjoint partitions for **R** (respectively, **P**) of roughly equal sizes by hashing on the join values for location (this is reminiscent of HyperCube[14] and Shares [1] algorithms for join-aware data partitioning across servers). By joining each



(a) Size and join time.



(b) Learning time.



(c) Compression ratio and speedup.

Figure 5.1: Housing dataset: Size and join time for factorised vs. flat joins, and learning time for the three systems (absolute and relative). The learning speedup is w.r.t. **R**, which is faster than **P** in these experiments. **P** and **R** run out of memory already at $s = 17$ for join and $s = 11$ for learning.

partition with the other tables we obtain a partitioning of the entire join result. The (starred) time to compute the join for all four partitions is reported in Table 5.1.

5.2.3 Importing Datasets

F is tightly integrated with the FDB query engine, so there is no time cost associated with the import of the training dataset. **P** and **R**, however, need one pass over the join result to load it into their internal data structures. This is typical for the existing solutions based on software enterprise stacks consisting of dozens of specialised systems (e.g., for analytics, OLAP, and OLTP, and BI), where non-trivial integration effort is usually spent at the interface between these systems. Table 5.1 and Figure 5.2 report the times for importing the training dataset constructed from the real-world datasets and respectively the synthetic dataset. For the Housing dataset, **P** and **R** failed to import the data starting with $s = 10$ and respectively $s = 11$. In contrast, **F** can finish even for $s = 100$.

5.2.4 Learning without Feature Interactions

Table 5.1 and Figure 5.4(b) show the performance of learning tasks with the three systems. For all datasets, the speedup of learning with **F** vs. the competitors closely follows the compression ratio (up to three orders of magnitude), as expected. For LastFM, it can even exceed it when compared to **R**. We noticed in this experiment only that **R**, does much more IO reads than **P**, so it spends significantly more time to load parts of the training dataset than the other two systems. **F** and **P** show a similar IO behaviour in our experiments. For the synthetic dataset, Figure 5.4(b) reports the performance of learning for up to scale 20 for **F** and up to 10 for **P** and **R**. The largest scale factor reported in this figure is 20 (compression ratio 1.9K, **F** takes 3.2 seconds for learning) and the largest one for which **R** works is 10 (compression ratio 240, **F** takes 2.1 seconds for learning).

For **R** and **P** on US retailer, we partitioned the dataset as explained in section 5.2.2. Table 5.1 reports the sum of learning times for all partitions. However, the learned parameters for each partition are arbitrarily far from true ones (although not supported by **P** and **R**, they could have been made correct if the output parameters for a partition would serve as the initial parameter values for the next partition).

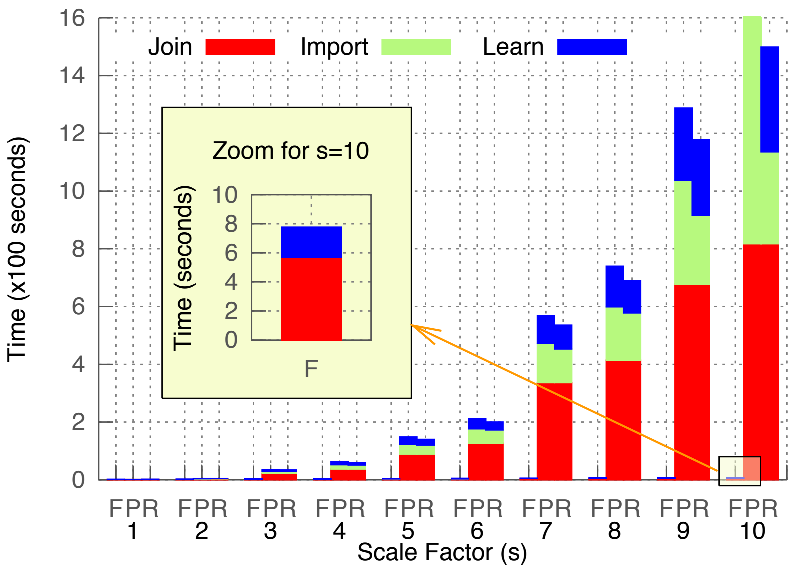


Figure 5.2: Housing dataset: Total time for the end-to-end solution (join, import, learn) for F, P, and R. **P** and **R** run out of memory starting with $s = 10$ and $s = 11$, respectively.

Nevertheless, even under these simplifying assumptions for **P** and **R**, **F** learns the correct parameters almost two orders of magnitude faster than its competitors.

5.2.5 Learning with Feature Interactions

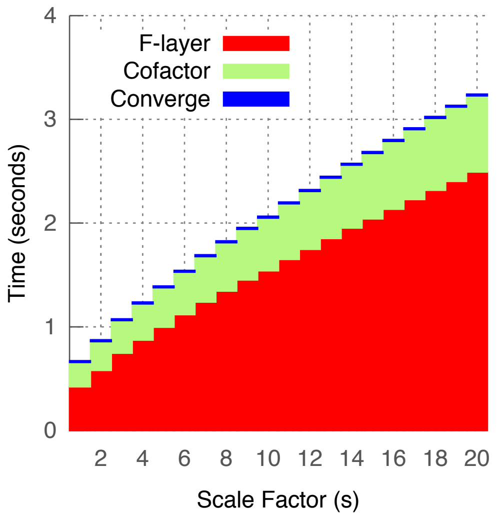
We report the times for tasks N_1 and N_2 with feature interactions on US retailer in Table 5.1. While the join remains the same, the time to perform the regression task slightly increases, as expected. Moreover, Figure 5.3(b) confirms a similar behaviour for the tasks N_1 and N_2 on Housing. We further observed that the influence on performance of basis functions defined on a single feature is very marginal.

5.2.6 Breakdown of Learning Time for F

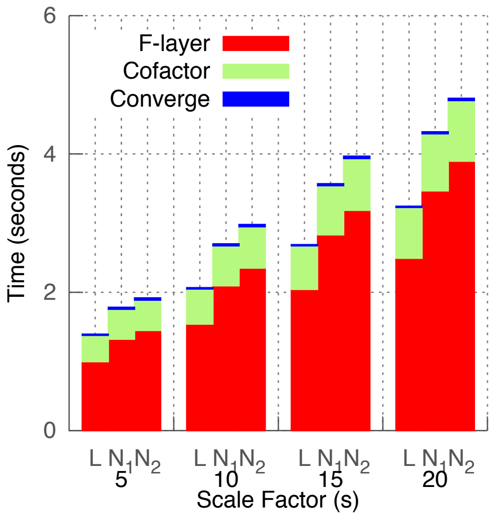
Figure 5.3(a) confirms the expected linear-time behaviour of all **F** components. The first component, which is the F-layer construction, takes the lion's share of computation time since it creates one node in the F-layer for each product and union node in the factorisation. We decoupled the times for computing counts, weighted sums, and cofactors, and found that their cost is about the same. The convergence of the parameters takes the least time and regardless of the scale factor, since it does not depend on the size of the data. It needs up to 2K steps to converge the values of parameters using the AdaGrad [19] continuous adaptation of the learn rate. In case of feature interactions, we observe that the time slightly increases for each component, cf. Figure 5.3(b).

5.2.7 End-to-End Solution

We also report the performance of end-to-end solutions using **F** based on the FDB query engine and using **P** and **R** based on RDB. Table 5.1 shows this as total time



(a) **F** without interactions.



(b) **F** with interactions.

Figure 5.3: Housing dataset: Breakdown of **F**'s timing without interactions (a) and with interactions (b).

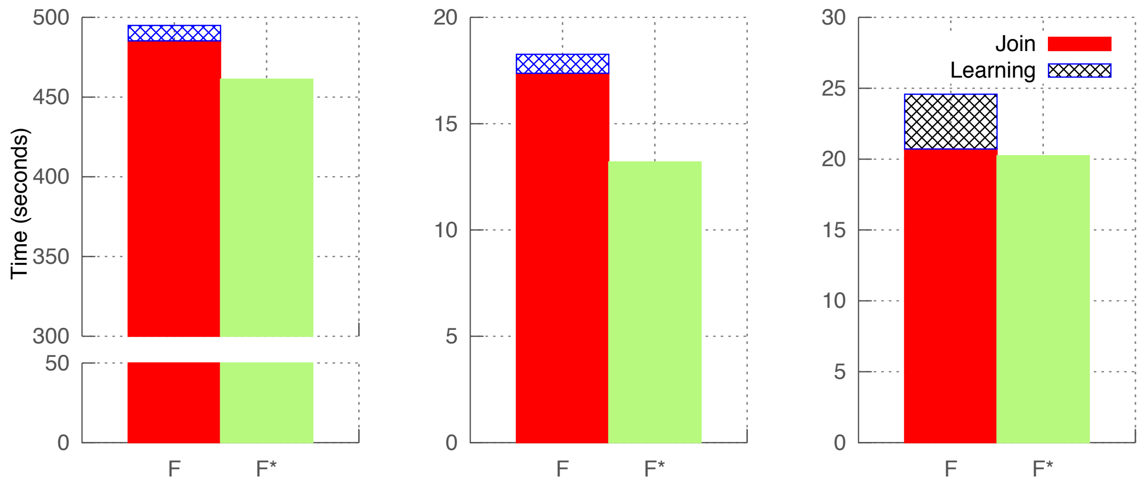
for real datasets, while Figure 5.2 shows it for the synthetic dataset. The speedup is in most cases higher than for join processing and lower than for learning. We stress-tested **F** for scale factors beyond 20 for the synthetic dataset: for $s = 50$ (compression ratio 14K), the total time is of 6.3 seconds, while for $s = 100$ (compression ratio 69K), it takes 11.4 seconds.

5.2.8 Model selection

In the previous experiments, we learned over all features of the training dataset. We further considered settings with fewer features, as used for model selection. The experiments validated that **F** only computes the parameter cofactors once and that the convergence time is consistently the smallest amongst all components of **F**. This contrasts with **P** and **R** that need to independently learn over the entire dataset for each set of features.

5.2.9 Comparing F and F*

We present an empirical comparison between **F** and **F*** for three real world datasets (US Retailer, LastFM L2, and MovieLens) in Figure 5.4. The actual data and the gain in comparison to **R** and **P** is provided in Figure 5.1. The plots show that the end-to-end solution of **F*** can outperform the combination of **F** and FDB, if the input is given in relation form and no modification of the factorised join is required. In particular, the end-to-end solution provided by **F***, which builds the factorisation and then learns the model, outperforms the factorised join computation of FDB alone. The total performance gain can be attributed to two things: (a) by merging the F-layer and factorised join computations, we do no longer have to create the F-layer on top of the factorisation and, therefore, avoid object creations; (b) the factorisation used for **F*** has been scaled down by only creating one node for all siblings at the leaf level, which saves time and memory space.



(a) US Retailer. (b) LastFM L2. (c) MovieLens.

Figure 5.4: Performance comparison between the end-to-end solution for **F** (including the factorised join computation) and **F*** for three real datasets.

For the other datasets, we can also see performance improvements, but in a much smaller proportion. For LastFM L1, the small gap in the performance (from 6.19 seconds to 6.06 seconds) can be attributed to its size. The dataset is comparatively small and, therefore, no big performance differences can be expected when comparing **F** and **F***.

The housing dataset, we realise the following performance gaps for selected scaling factors:

SF	**F***	**F**
5	4.5	5.2
10	6.8	7.8
20	13.6	15.5
50	31.0	32.2
100	59.5	61.6

Table 5.2: Comparison of **F*** and end-to-end solution for **F** (which includes the factorised join computation) for Housing dataset with selected scaling factors.

The reason for this rather limited performance improvement for the housing dataset is the way that this synthetic dataset has been created. The factorised join does have almost no possible factorisations past the join attributes, which means that at the leaf level there will only be one node for each branch. Consequently, the algorithm cannot benefit from the optimisation that all sibling nodes at the leaf level are combined into one node. The regression model does not benefit from the potential performance improvements; future work will investigate how the join computation for **F*** can be scaled down further to provide performance improvements for all kinds of datasets, including the housing dataset.

5.2.10 Further findings

For the sake of comparing the performance of F with another iterative learning algorithm, we have used the R glmnet [21] package, which implements linear regression with coordinate descent optimisation. Contrary to F, glmnet approximates the model with regularisers, which is why we refrained from using this package for full benchmarking. Nevertheless, our experiments show that F is 11 times faster for the housing dataset with scaling factor 10 and 6 times faster for US retailer with 4 partitions. This is significant because glmnet is one of the fastest packages for learning generalised linear models.

We evaluated the three systems on two further real-world datasets: (1) the *Delicious* dataset that is provided together with LastFM [12] and (2) the *Financial* dataset [7], traditionally used for regression tasks. The Delicious dataset behaved very similarly to LastFM for the same types of joins. The Financial dataset is very small relative to the others and all systems' learned time was under one second.

Chapter 6

Related work

Our contribution lies at the interface between *databases* and *machine learning*, as we look at regression learning, which is a fundamental machine learning problem, through database glasses. The crossover between these two communities gained increasing interest during the past years, as also acknowledged in a SIGMOD 2015 panel [40]. We position our contribution in the realm of learning tasks backed up by database techniques and then highlight related, yet orthogonal to ours, work from the machine learning community on optimising the performance of the gradient descent.

6.1 Machine learning in databases

Our work follows a very recent line of research on marrying databases and machine learning, e.g., [40, 16, 43, 27, 2, 32, 23, 11, 45, 9].

Two of these works are closest in spirit to ours since they investigate the impact of data factorisation for the purpose of boosting learning tasks. Rendle [43] considers a limited form of factorisation of the design matrix for regression. His approach performs ad-hoc discovery of repeating patterns on the flat join to save up time for the subsequent regression task, though with two important limitations: This discovery cannot capture join dependencies, i.e., recover the knowledge that the data has been produced via joins, since this is NP-hard; it does not save time for computing the join, but it instead needs additional time to perform the discovery. Our approach is different since (i) we avoid the computation of the flat join as it is too expensive and entails redundancy; (ii) we exploit the join structure from the query to identify the repeating patterns due to join dependencies. Kumar et al. [27] propose algorithms for learning over one key-foreign key join and consider factorised computation over the non-materialised join. In contrast, our approach works for arbitrary join queries and factorisations with caching, linear regression with feature interactions, and has theoretical guarantees.

Most efforts in the database community are on designing systems to support large-scale scalable machine learning on top of distributed architectures such as Spark [51], e.g., MLLib [2], DeepDist [32], SystemML [23, 9], system benchmarking [11] and sample generator for cross-validate learning [45]. Our approach focuses on linear regression and pushes the performance barrier in the one-machine scenario. It achieves this by factorised data and computation, which enables more data to be kept in the main memory of one machine and to be processed very fast without the need for distribution. As pointed out recently [29], when benchmarked against one-machine systems, distributed systems can have a non-trivial upfront cost that can be offset by more expensive hardware or large problem settings. An interesting direction of research is to integrate our system **F** into one of the existing distributed architectures such as Spark.

The tight integration of the FDB query engine for factorised databases [6] with our regression learner **F** has been inspired by LogicBlox [3], which provides a unified runtime for the enterprise technology stack.

6.2 Gradient descent optimisation

The gradient descent family of optimisation algorithms is fundamental to machine learning and very popular, cf. a ICML 2015 tutorial [44]. One of the applications of gradient descent is regression, which is the focus of this paper. A popular variant is the stochastic gradient descent [10], which takes a gradient descent step based on one training example instead of the entire training dataset. Although individual convergence steps can be inaccurate, the algorithm is used in practice because it scales well for large amounts of data. For this reason, stochastic gradient descent has attracted a lot of interest in the academic community and consequently a series of improvements have been proposed: (i) improvements on the convergence rate via adaptive learning rate [19] (which is also used by our systems **F** and **F***); and (ii) parallel or distributed versions [52, 38, 41, 17, 35]. Some of these optimisations have made their way in systems such as DeepDive [46, 28] and DeepDist [32, 17]. Our contribution is orthogonal since it focuses on avoiding data and computation redundancy in the special case of learning over joins.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this dissertation, we put forward **F**, a fast learner of least squares regression models over arbitrary joins of input relations. Current state-of-the-art analytics systems, such as R and Python StatsModels, require a flat relation input and are not integrated in a database management system. This leads to unnecessary redundancy for both computation and data representation that is not required for learning regression models.

By exploiting the theoretical and practical guarantees of factorised join representations, we are able to reduce redundancy in the input data and achieve a better join performance. Furthermore, we use an algebraic rewriting of the least squares objective function, which allows us to decouple the computation of the parameter cofactors and the optimisation procedure. This property enables us to learn a parameterised least squares regression model with only two passes over the factorisation. **F** outperforms R and Python StatsModels by up to three orders of magnitude on a variety of real-world and synthetic datasets.

Comparable state-of-the-art analytics systems, which include R and Python Stats Models, are separate from the database management system that computes the join, which means that additional time is spent on loading the information from the database to the analytics system. Our system **F** is integrated in the query engine FDB, which eliminates the loading time of the data and enables the immediate processing of the join.

In line with this observation, we propose an optimised version, called **F***, which is a stand-alone, end-to-end system that merges the computation for learning with the computations required for the factorised join. This system provides additional performance improvements over **F**, because the factorisation can be structured to be optimal for learning regression models. The performance improvements come with the limitation that **F*** can only take the data directly for the input relations and cannot perform any modifications at the join output level.

This is the first work that exploits factorised joins for machine learning algorithms and, therefore, there is a lot of potential for future research, some of which we outline in the final section.

7.2 Future Work

The future research that follows from the work that has been conducted in preparation for this dissertation is two-fold. On the one hand, we are aiming to extend and improve the proposed system. On the other hand, we investigate how other machine learning algorithms can benefit from factorised data, similar to the system proposed in this dissertation. We will briefly outline some of the findings we have already come across, but were not able to pursue yet.

7.2.1 Extending F and F*

Distribution

One main objective is to distribute the computations performed by **F** and **F*** on multiple cores and ultimately multiple machines. In Proposition 3.1 we showed that cofactor computation commutes over unions, which implies that the cofactors for the entire factorisation can be computed concurrently on multiple fragments of the factorisation. It remains to be investigated how the factorisation can be distributed so that it provides the most performance improvements for the learning process.

Optimising F-layer construction for F*

As mentioned in Section 4.2, it is possible for **F*** to scale down the factorisation that is computed in combination with the F-layer. This optimisation will require a modification of the inherent structure of the factorisation, but it will also ensure that the maximum performance gain can be achieved by the proposed end-to-end regression learner.

Regularisation

Besides improving the current system from an implementation standpoint, we also investigate how the regression learner can be extended with state-of-the-art machine learning techniques. This includes the regulariser, which is commonly used to avoid the problem of overfitting. The regulariser limits the size of the parameters of the model by creating penalty for too large parameters. Ultimately, the parameters are forced to stay close to zero, which implies that the model cannot become too complex and overfit on the training data.

Learning over Aggregates

In Example 3.10, we show that it can be necessary to perform aggregates over the input data to attain the desired features for a model. In future research, we would like to investigate this problem further to understand how aggregates can be supported most effectively for the context of learning. We will also investigate how aggregates can be integrated in **F** which is in line with the general idea of integrating machine learning in databases.

Push F over join

The major impediment for immediate adoption of **F** in existing commercial systems is that it works with factorisations instead of relations. Based on the findings in our research, it may be possible to compute the count and weighted sums that constitute the cofactor matrix, the major component of **F**, directly on the input relations. This would eliminate the construction of the F-layer and would provide additional performance improvements.

More succinct factorisations

F is robust in the sense that it would work even for more compact factorisations. While we show it for factorisations with caching, it even works for factorisations with caching, where different branches have different nesting structures. Such adaptive factorisations can bring more succinctness. In future research, we plan to look into how to create such adaptive factorisations for join results and how they can be exploited for learning.

7.2.2 Extending the class of algorithms that benefit from factorisations

Besides improving the proposed regression learner, we aim to extend our approach to a family of machine learning techniques. The goal is to provide a class of learning models that exploit factorised data so that they can be solved in constant number of passes over the data. The additional techniques will be applied to a range of applications outside of the realm of regression tasks, such as clustering or classification problems. Some of the algorithms that we will consider in the near future are:

Factorisation Machines

The Factorisation Machine [42] is a general predictor for both regression and classification tasks. Therefore, it is similar to the Support Vector Machines but it provides the advantage that it can also be applied to data with high sparsity. The underlying framework of factorisation machines relies on a linear prediction model in combination with variable interaction terms for all features. Therefore, the main framework depends on the components for which we have shown that they can benefit from factorised data representation. For this reason, we are positive that factorisation machines in general can also benefit from factorised data representation.

k-Nearest-Neighbour

The k-Nearest-Neighbour (k-NN) algorithm [8] is an instance-based machine learning algorithm that can be applied for both regression and classification tasks. This means that the algorithm does not learn a parameterised model, but instead bases the prediction solely on a given training dataset. For a new input feature vector, k-NN finds the k training samples that are closest to the this new feature. The output is then the average of the labels of the k training samples (for regression) or the majority vote over the categories of the k training samples (for classification).

Since the factorised data representations can be exponentially smaller in size than the equivalent flat representation of a dataset, a modified k-NN algorithm that exploits the factorisation can provide significant performance improvements in comparison to the vanilla approach. We have implemented an algorithm that performs k-NN in one bottom-up pass over the factorisation, but due to time constraints we have not been able to benchmark our approach and, therefore, cannot present the results in this dissertation. Future research will investigate this problem and add k-NN to suit of algorithms that benefit from factorised data.

Classification and Regression Trees with Boosting

A method that is often used in practice are ensembles of regression trees. These trees consist of simple decision rules, similar to a decision tree, which predict a score at each leaf of the tree. Each tree splits the feature space into segments and for each segment it gives a prediction for the output. Although one tree only gives a crude estimate of the output, it is possible to combine a series of multiple simple regression trees in one consistent ensemble. This idea of using combining many simple functions to create a complex model is the general framework of boosting.

Ensembles of regression trees are often used in industry because they scale well with large amounts of data, are easy to learn, and provide an simple framework to model interactions between features [13]. Furthermore, they have proven to be very effective predictive models, which is shown by their success for in various data mining competitions [25].

Since factorised representations of data are in essence parse trees, it is reasonable to believe that they provide an effective framework to build regression trees. Intuitively, the f-tree that defines the nesting for the natural join should provide a good structure for decision trees. This, however, will have to be investigated in future research. It will also be interesting to see how insights from regression tress will be applicable to **F** and other machine learning algorithms that are built on factorised data representations.

Bibliography

[1] Foto N. Afrati and Jeffrey D. Ullman. Optimizing multiway joins in a Map-Reduce environment. *IEEE TKDE*, 23(9):1282–1298, 2011.

[2] Apache. MLlib: Machine learning in Spark, `https://spark.apache.org/mllib`, 2015.

[3] 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.

[4] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. In *FOCS*, pages 739–748, 2008.

[5] Nurzhan Bakibayev, Tomás Kociský, Dan Olteanu, and Jakub Závodný. Aggregation and ordering in factorised databases. *PVLDB*, 6(14):1990–2001, 2013.

[6] Nurzhan Bakibayev, Dan Olteanu, and Jakub Závodný. FDB: A query engine for factorised relational databases. *PVLDB*, 5(11):1232–1243, 2012.

[7] Petr Berka. PKDD discovery challenge: A collaborative effort in knowledge discovery from databases, 1999, `http://lisp.vse.cz/pkdd99/Challenge/chall.htm`.

[8] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[9] 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.

[10] Léon Bottou. Stochastic gradient descent tricks. In *Neural Networks: Tricks of the Trade (2nd ed)*, pages 421–436, 2012.

[11] Zhuhua Cai, Zekai J. Gao, Shangyu Luo, Luis Leopoldo Perez, Zografoula Vagena, and Christopher M. Jermaine. A comparison of platforms for implelementing and running very large scale machine lear- ning algorithms. In *SIGMOD*, pages 1371–1382, 2014.

[12] Iván Cantador, Peter Brusilovsky, and Tsvi Kuflik. 2nd workshop on information heterogeneity and fusion in recommender systems. In *RecSys*, pages 387–388, 2011, `http://grouplens.org/datasets/hetrec-2011`.

[13] Tianqi Chen. *Introduction to Boosted Trees.* University of Washington, `https://homes.cs.washington.edu/~tqchen/pdf/BoostedTree.pdf`, 2014.

[14] Shumo Chu, Magdalena Balazinska, and Dan Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In *SIGMOD*, pages 63–78, 2015.

[15] Radu Ciucanu, Dan Oltenau, and Maximilian Schleich. Learning Linear Regression Models over Factorized Joins. Submitted to *SIGMOD* 2016.

[16] Tyson Condie, Paul Mineiro, Neoklis Polyzotis, and Markus Weimer. Machine learning for big data. In *SIGMOD*, pages 939–942, 2013.

[17] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew W. Senior, Paul A. Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *NIPS*, pages 1232–1240, 2012.

[18] Jack Dongarra and Francis Sullivan. Guest editors' introduction: The top 10 algorithms. *Computing in Science & Engineering*, 2(1):22–23, 2000.

[19] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *JMLR*, 12:2121–2159, 2011.

[20] J. G. F. Francis. The QR transformation: A unitary analogue to the LR transformation–Part 1. *The Computer Journal*, 4(3):265–271, 1961.

[21] Jerome H. Friedman, Trevor Hastie, and Rob Tibshirani. Regularization paths for generalized linear models via coordinate descent. *Journal of Statistical Software*, 33(1):1–22, 2 2010.

[22] GroupLens Research. MovieLens, `http://grouplens.org/datasets/movielens`, 2003.

[23] 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.

[24] J. Jaccard, R. Turrisi, and C.K. Wan. *Interaction Effects in Multiple Regression.* Number Bd. 72;Bd. 1990 in Interaction Effects in Multiple Regression. SAGE Publications, 1990.

[25] Kaggle. *Random Forests.* `https://www.kaggle.com/wiki/RandomForests`, 2015.

[26] Mahmoud Abo Khamis, Hung Q. Ngo, Christopher Ré, and Atri Rudra. FAQ: Questions asked frequently, CoRR:1504.04044, 2015.

[27] Arun Kumar, Jeffrey F. Naughton, and Jignesh M. Patel. Learning generalized linear models over normalized data. In *SIGMOD*, pages 1969–1984, 2015.

[28] Ji Liu, Steve Wright, Christopher Ré, Victor Bittorf, and Srikrishna Sridhar. An asynchronous parallel stochastic coordinate descent algorithm. In *ICML*, pages 469–477, 2014.

[29] Frank McSherry, Michael Isard, and Derek Gordon Murray. Scalability! but at what COST? In *HotOS*, 2015.

[30] Ronald Menich and Nik Vasiloglou. The future of LogicBlox machine learning. LogicBlox User Days, 2013.

[31] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective.* The MIT Press, 2012.

[32] Dirk Neumann. Lightning-fast deep learning on Spark via parallel stochastic gradient updates, `www.deepdist.com`, 2015.

[33] Andres Ng. *CS229 Lecture Notes.* Stanford & Coursera, `http://cs229.stanford.edu/`, 2014.

[34] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. In *PODS*, pages 37–48, 2012.

[35] Feng Niu, Benjamin Recht, Christopher Ré, and Stephen J. Wright. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *In NIPS*, 2011.

[36] Dan Olteanu and Jakub Závodný. Size bounds for factorised representations of query results. *TODS*, 40(1):2, 2015.

[37] Roger Penrose. A generalized inverse for matrices. *Math. proc. of the Cambridge phil. soc.*, 51(03):406–413, 1955.

[38] Fabio Petroni and Leonardo Querzoni. GASGD: stochastic gradient descent for distributed asynchronous matrix completion via graph partitioning. In *RecSys*, pages 241–248, 2014.

[39] R Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, `www.r-project.org`, 2013.

[40] Christopher Ré, Divy Agrawal, Magdalena Balazinska, Michael I. Cafarella, Michael I. Jordan, Tim Kraska, and Raghu Ramakrishnan. Machine learning and databases: The sound of things to come or a cacophony of hype? In *SIGMOD*, pages 283–284, 2015.

[41] Benjamin Recht and Christopher Ré. Parallel stochastic gradient algorithms for large-scale matrix completion. *Math. Program. Comput.*, 5(2):201–226, 2013.

[42] Steffen Rendle. Factorization machines with libfm. *ACM Trans. Intell. Syst. Technol.*, 3(3):57:1–57:22, May 2012.

[43] Steffen Rendle. Scaling factorization machines to relational data. *PVLDB*, 6(5):337–348, 2013.

[44] Peter Richtárik and Mark Schmidt. Modern convex optimization methods for large-scale empirical risk minimization. In *ICML*, 2015. Invited Tutorial.

[45] Sebastian Schelter, Juan Soto, Volker Markl, Douglas Burdick, Berthold Reinwald, and Alexandre V. Evfimievski. Efficient sample generation for scalable meta learning. In *ICDE*, pages 1191–1202, 2015.

[46] Jaeho Shin, Sen Wu, Feiran Wang, Christopher De Sa, Ce Zhang, and Christopher Ré. Incremental knowledge base construction using DeepDive. *PVLDB*, 8(11):1310–1321, 2015.

[47] G.W. Stewart. *Matrix Algorithms: Volume 1: Basic Decompositions*, chapter The QR Decomposition and Least Squares, pages 251–358. Society for Industrial and Applied Mathematics, 1998.

[48] Gilbert Strang. *Introduction to Linear Algebra*. Wesley-Cambridge Press, 2003.

[49] The StatsModels development team. StatsModels: Statistics in Python, `http://statsmodels.sourceforge.net`, 2012.

[50] Todd L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In *ICDT*, pages 96–106, 2014.

[51] 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 15–28, 2012.

[52] Martin Zinkevich, Markus Weimer, Alexander J. Smola, and Lihong Li. Parallelized stochastic gradient descent. In *NIPS*, pages 2595–2603, 2010.

Appendix

Proofs

Proof of Proposition 3.1

1. (*Symmetry*). Assume that $Q(\mathbf{D})$ has m tuples. We have:

$$S_k = \sum_{i=1}^{m} (\theta_0 x_0^{(i)} + \ldots + \theta_j x_j^{(i)} + \ldots + \theta_n x_n^{(i)}) x_k^{(i)},$$
$$S_j = \sum_{i=1}^{m} (\theta_0 x_0^{(i)} + \ldots + \theta_k x_k^{(i)} + \ldots + \theta_n x_n^{(i)}) x_j^{(i)}.$$

This implies that:

$$\text{Cofactor}[A_j, A_k] = \sum_{i=1}^{m} x_j^{(i)} x_k^{(i)} = \sum_{i=1}^{m} x_k^{(i)} x_j^{(i)} = \text{Cofactor}[A_k, A_j].$$

2. (*Commutativity with union*). For $1 \leq l \leq p$, assume that the training dataset $Q(\mathbf{D}_l)$ has m_l tuples, that we denote

$$Q(\mathbf{D}_l) = \{(x_{l,0}^{(1)}, \ldots, x_{l,n}^{(1)}), \ldots, (x_{l,0}^{(m_l)}, \ldots, x_{l,n}^{(m_l)})\}.$$

Then, $Q(\mathbf{D})$ has totally $\sum_{l=1}^{p} m_l$ tuples. Take $0 \leq k, j \leq n$. It holds that:

$$\begin{aligned}\sum_{l=1}^{p} \text{Cofactor}_l[A_k, A_j] &= \sum_{l=1}^{p} (\sum_{i_l=1}^{m_l} x_{l,k}^{(i_l)} x_{l,j}^{(i_l)}) \\ &= \sum_{i=1}^{\sum_{l=1}^{p} m_l} x_k^{(i)} x_j^{(i)} = \text{Cofactor}[A_k, A_j].\end{aligned}$$

3. (*Commutativity with projection*). Assume that $Q(\mathbf{D})$ has m tuples. Since we are under bag semantics, $\pi_L(Q(\mathbf{D}))$ also has m tuples. We assume that L has n_L features that we denote $x_{L,0}, \ldots, x_{L,n}$. Take $1 \leq k, j \leq n_L$. Then, $\text{Cofactor}_L[A_k, A_j] = \sum_{i=1}^{m} x_j^{(i)} x_k^{(i)}$, which moreover, is equal to $\text{Cofactor}[A_k, A_j]$.

Dataset & Task	$s(F)$	$s(Q)$	*fhtw*(Q)	$\rho^*(Q)$
Housing L	1 *	1	1	4
Housing N_1				
Housing N_2	2			
US retailer L	2 *	2	1	3
US retailer N_1				
US retailer N_2				
LastFM L_1	1 *	1	1	2
LastFM L_2				4
MovieLens L	2 *	2	1	3

Table 1: Details about the considered f-trees. We add "*" to the cases of asymptotically optimal f-trees. Although Housing has 6 relations and each of them has at least one attribute that does not occur in the others, we have $\rho^*(Q) = 4$ because `postcode` is key for `Transport` and `Demographics` hence these two relations do not contribute to $\rho^*(Q)$.

Proof of Theorem 3.8.

Take the basis functions $\phi_0, \ldots, \phi_b$ over the sets of features $S_0, \ldots, S_b$, the induced relational schema $\sigma = (R_0(S_0), \ldots, R_b(S_b))$, and $Q_\sigma = Q \bowtie R_0 \bowtie \cdots \bowtie R_b$ (the extension of Q w.r.t. σ).

First, we show that the extension $\mathbf{D}_\sigma$ of $\mathbf{D}$ with relations over the induced relational schema σ leads to a factorisation of the join $Q_\sigma(\mathbf{D}_\sigma)$ of size $O(|\mathbf{D}|^{\mathit{fhtw}(Q_\sigma)})$. The extension $\mathbf{D}_\sigma$ has a relation instance R_i (for $1 \leq i \leq b$) for each schema $R_i(S_i)$ that is the projection of $Q(\mathbf{D})$ on S_i i.e., $\pi_{S_i}(Q(\mathbf{D}))$. It then holds that $Q(\mathbf{D}) = Q(\mathbf{D}_\sigma) = Q_\sigma(\mathbf{D}_\sigma)$. By Proposition 2.2, there exists a factorisation of $\mathbf{D}_\sigma$ of size $O(|\mathbf{D}_\sigma|^{\mathit{fhtw}(Q_\sigma)})$. Under data complexity, the schema σ has constant size and thus $O(|\mathbf{D}_\sigma|^{\mathit{fhtw}(Q_\sigma)}) = O(|\mathbf{D}|^{\mathit{fhtw}(Q_\sigma)})$. Let us denote by E the factorisation of $Q_\sigma(\mathbf{D}_\sigma)$ and let F be an f-tree under which we obtain the size of $O(|\mathbf{D}|^{\mathit{fhtw}(Q_\sigma)})$.

We next show that E can be augmented with singletons corresponding to the interaction terms ϕ_k while keeping the same asymptotic bound on its size. Since F satisfies the path constraints, the attributes of any relation, and in particular of relations over schemas $R_0(S_0), \ldots, R_b(S_b)$, are along the same root-to-leaf path in F. This means that E materialises all possible combinations of singletons for attributes in each schema S_k. We can thus compute the result r_k of the basis function ϕ_k on the singletons for attributes in S_k in one pass over E. For each tuple of singletons attributes in S_k, we add a product with the singleton $\langle \phi_k : r_k \rangle$ immediately under the lowest singleton in the tuple in E. We also add a node ϕ_k to F immediately under the lowest attribute in S_k. These additions do not modify the asymptotic size bounds of E since we add one singleton for each S_k-tuple of singletons in E. Let us denote by E_σ the factorisation E extended with singletons for interaction terms.

The factorisation E_σ has size $O(|\mathbf{D}|^{\mathit{fhtw}(Q_\sigma)})$ and singletons for each basis function ϕ_k. Learning over E_σ has thus reduced to learning with identity basis functions and, by Proposition 3.2, it can be done in two passes over E_σ. The claim follows.

- `House (postcode, livingarea, price, nbbedrooms, nbbathrooms, kitchensize, house, flat, bungalow, garden, parking)`
- `Shop (postcode, openinghoursshop, pricerangeshop, sainsburys, tesco, ms)`
- `Institution (postcode, typeeducation, sizeinstitution)`
- `Restaurant (postcode, openinghoursrest, pricerangerest)`
- `Demographics (postcode, averagesalary, crimesperyear, unemployment, nbhospitals)`
- `Transport (postcode, nbbuslines, nbtrainstations, distancecitycentre)`

(a) Schema for Housing.

- `Inventory (locn, date, ksn, inventoryunits)`
- `Location (locn, zip, d1, ..., d10)`
- `Census (zip, population, white, asian, pacific, blackafrican, medianage, occupiedhouseunits, houseunits, families, households, husbwife, males, females, householdschildren, hispanic, state)`

(b) Schema for US retailer.

- `Userfriends (user, friend)`
- `Userartists (user, artist, weight)`
- `Usertaggedartiststimestamps (user, artist, tag, timestamp)`

(c) Schema for LastFM.

- `Ratings (user, movie, rating, timestamp)`
- `Users (user, age, gender, occupation, zipcode)`
- `Movies (movie, year, action, adventure, animation, children, comedy, crime, documentary, drama, fantasy, filmnoir, horror, musical, mysery, romance, scifi, thriller, war, western)`

(d) Schema for MovieLens.

Figure 1: Schemas of the considered datasets.

More details about datasets

We present the schemas for all considered datasets in Figure 1. For all datasets, the values are of type double. We depict the f-trees for all considered regression tasks in Figure 2 and we provide some asymptotic details on them in Table 1. Next, we detail each of the datasets.

• **Housing.** The schema of the Housing dataset consists of 6 relations `House`, `Shop`, `Institution`, `Restaurant`, `Demographics`, `Transport` over a total of 27 attributes, that we detail in Figure 1(a). We consider the natural join (on `postcode`). To construct instances of the Housing dataset, we randomly generated numerical values for each attribute. The domains are intervals simulating the real-world semantics of attributes e.g., large intervals for attributes such as `price`, smaller intervals for attributes such as `nbtrainstations` and Boolean values for attributes such as `house`, `flat`, or `bungalow` indicating the type of housing.

Linear task L. Since all relations have a common attribute (`postcode`), the above query is *hierarchical*, and consequently, $s(Q) = 1$. Then, the f-tree that we consider is optimal i.e., has $s(F) = 1$, which intuitively means that each relation corresponds to a root-to-leaf path in the f-tree. We present a snapshot of it in Figure 2(a).

Linear task with basis functions N_1. We recall that N_1 considers two interactions: (i) between the type of house and the number of bedrooms (both features of `House`), and (ii) between the number of train stations and the distance to city center (both features of `Transport`)[1]. Since both features occurring in a same interaction belong to the same relation, we can use precisely the same f-tree from the linear case, which is asymptotically optimal.

Linear task with basis functions N_2. We recall that N_2 considers two interactions: (i) between the number of bus lines (feature of `Transport`) and the shop opening hours (feature of `Shop`), and (ii) between the size of educational institutions (feature of `Institution`) and the number of crimes per year (feature of `Demographics`)[2]. Both interactions from N_2 consider pairs of features from different relations, which implies that the f-tree that we use for this task must satisfy two additional constraints: `nbbuslines` and `openinghoursshop` should be on the same root-to-leaf path, and moreover, `sizeinstitution` and `crimesperyear` should be on the same root-to-leaf path. Consequently, the obtained f-tree has $s(F) = 2$ that is not asymptotically optimal since $s(Q)$ remains 1. We depict a snapshot of the considered f-tree in Figure 2(b).

• **US retailer.** The schema of the US retailer dataset consists of 3 relations `Inventory`, `Census`, and `Location` over a total of 31 attributes, that we detail in Figure 1(b). By `d1` to `d10` we denote the distances between the respective location and several stores. The considered query is the natural join (on `locn` and `zip`).

Linear task L. The above query has $s(Q) = 2$ and we consider an asymptotically optimal f-tree (i.e., $s(F) = 2$). We present a snapshot of this f-tree in Figure 2(d). The relation `Inventory` is encoded along the path `locn/ksn/...`, the relation `Location` is encoded along the path `locn/zip/d1/...`, while `Census` is encoded along the path `zip/population/...`

Linear task with basis functions N_1. We recall that N_1 considers two interactions: (i) between the median age and the number of families, and (ii) between different distances to other stores (take wlog `d1` and `d2`)[3]. Since both features occurring in a same interaction belong to the same relation, we can use precisely the same f-tree from the linear case, which is asymptotically optimal.

Linear task with basis functions N_2. We recall that N_2 considers two interactions: (i) between the population and the number of house units, and (ii) between the median age and distance to another store[4]. While the features from the first interaction occur in the same relation `Census`, it is not the case for the second one since the median age

[1] The first interaction looks at the effect of having more bedrooms in a house as opposed to other types of accommodation, while the second one looks at identifying whether a good public transportation system could potentially impact the price if the housing is not close to the city center.

[2] The first interaction looks at the effect of the general availability of services on the housing price (used to identify where it is more beneficial to have a better public transport system or longer opening hours), while the second one considers another social aspect that could affect the price of the housing (indeed, both education and safety are two strong indicators for the quality of the district).

[3] The first interaction looks at the effect on inventory units while correlating the number of families and the median age, since the two features are strongly related. The second interaction looks at the effect of having competitors close to the store and how the interaction of the two competitors changes the effect on the inventory units in the given store.

[4] The first interaction uses the insight that if we have a large population but few houses, then many people live in the same house. This can give an indication of what the general population is

is a feature of `Census` and the distance to another store (take wlog `d1`) is a feature of `Location`. Thus, the f-tree used for N_2 needs to additionally satisfy the constraint that the features from its second interaction are on the same root-to-leaf path. We present a snapshot of it in Figure 2(e). However, notice that this f-tree is still asymptotically optimal i.e., has $s(F) = s(Q) = 2$. Indeed, the path `locn/zip/d1/medianage/...` can be covered by two relations (`Census` and `Location`), while each of the other paths can be covered by one relation (`Location` and `Inventory`, respectively). In such a case, contrarily to what observed for Housing dataset, imposing additional constraints due to interactions between features of different relations does not preclude the asymptotic optimality of the f-tree considered for a linear task with basis functions.

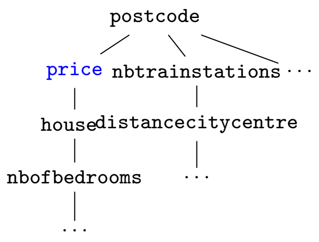
• **LastFM [12].** LastFM consists of 3 relations `Userfriends (UF), Userartists (UA), Usertaggedartiststimestamps (UTA)`, as detailed in Figure 1(c). We consider two queries, in both of them our regression task being to predict the weight for given user and artist, based on social networking information. In the first query L_1 (over 6 attributes), we consider as input only the friendship relation together with the weights for the users and friends, while in the second query L_2 (over 10 attributes), we additionally consider as input the tags and the timestamps for both users and friends. The precise queries are:

$$\begin{aligned} L_1 :&\ \texttt{UF} \bowtie_{\texttt{UF.user=UA1.user}} \texttt{UA1} \bowtie_{\texttt{UF.friend=UA2.user}} \texttt{UA2} \\ L_2 :&\ \texttt{UF} \bowtie_{\texttt{UF.user=UA1.user}} \texttt{UA1} \bowtie_{\texttt{UF.friend=UA2.user}} \texttt{UA2} \bowtie \\ &\ \bowtie_{\texttt{UF.user=UTA1.user}} \texttt{UTA1} \bowtie_{\texttt{UF.friend=UTA2.user}} \texttt{UTA2} \end{aligned}$$

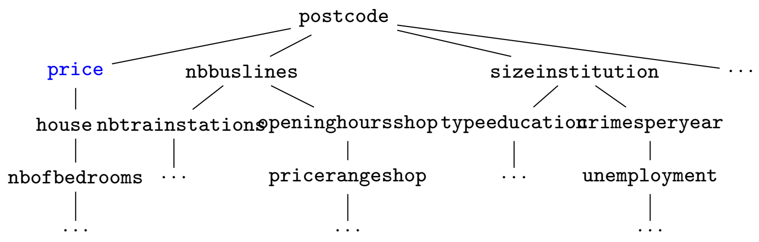
Both queries are hierarchical since the `user` is common to all joined relations. Hence, we can consider for both of them asymptotically optimal trees with $s(F) = 1$. We illustrate their f-trees in Figure 2(f) and Figure 2(g), respectively.

• **MovieLens [22].** MovieLens consists of 3 relations over 27 attributes, as detailed in Figure 1(d). We consider the natural join (on `user` and `movie`) and we depict a snapshot of the considered asymptotically optimal f-tree (with $s(F) = 2$) in Figure 2(c). There are four versions of the MovieLens datasets and we only reported our experimental findings for the largest available version (1M records) that has complete information for all three tables; there are two larger versions (10M and 20M) having a simplified schema without users. We also experimented with the other versions (100K and the larger ones where we synthetically generated the Users relation) and found that they exhibit the same compression ratio and relative performance gain.

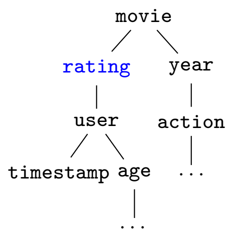
like, and therefore, can have a meaningful impact on our prediction. The second interaction looks at how the age of a person influences her tendency to look for satisfying stores at a further distance.



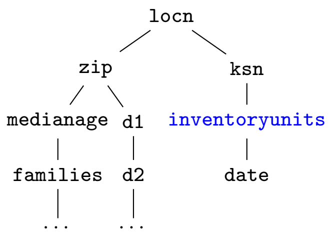
(a) Housing L and N_1.



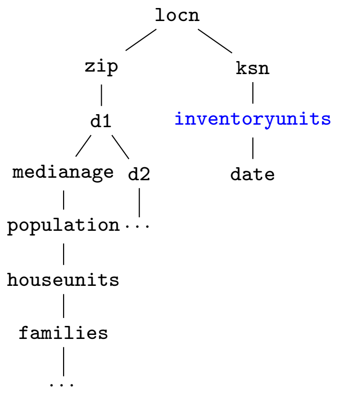
(b) Housing N_2.



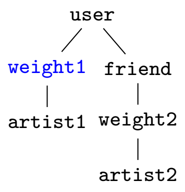
(c) MovieLens L.



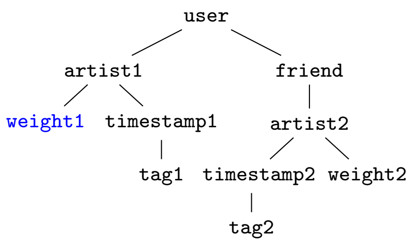
(d) US retailer L and N_1.



(e) US retailer N_2.



(f) LastFM L_1.



(g) LastFM L_2.

Figure 2: Snapshots of the f-trees considered for our experiments.

	θ_Z	θ_S	θ_P	θ_T	θ_H	θ_{SH}	θ_0
Σ_Z	$24z_1^2 + 2z_2^2$	$z_1S_5 + z_2S_8$	$z_1P_5 + z_2P_8$	$z_1T_5 + z_2T_8$	$z_1H_5 + z_2H_8$	$z_1SH_5 + z_2SH_8$	$24z_1 + 2z_2$
Σ_S	Σ_Z/θ_S	$18s_1^2 + 6s_2^2 + 2s_2^2$	$s_1P_2 + s_2P_4 + s_2P_7$	$s_1T_2 + s_2T_4 + s_2T_7$	$s_1H_2 + s_2h_4 + s_2h_7$	$s_1SH_2 + s_2SH_4 + s_2SH_7$	$18s_1 + 6s_2 + 6s_2$
Σ_P	Σ_Z/θ_P	Σ_S/θ_P	$9p_1^2 + 9p_2^2 + 6p_3^2 + 2p_4^2$	$3P_1T_1 + 3P_3T_3 + P_6T_3$	$3P_1H_1 + 2P_3H_3 + P_6H_6$	$3P_1SH_1 + 2P_3SH_3 + P_6SH_6$	$9p_1 + 9p_2 + 6p_3 + 2p_4$
Σ_T	Σ_Z/θ_T	Σ_S/θ_T	Σ_P/θ_T	$6(t_1^2 + t_2^2 + t_3^2) + 4(t_4^2 + t_5^2)$	$2T_1H_1 + T_3H_3 + T_3H_6$	$2T_1SH_1 + T_3SH_3 + T_3SH_6$	$3(t_1 + t_2 + t_3) + 4(t_4 + t_5)$
Σ_H	Σ_Z/θ_H	Σ_S/θ_H	Σ_P/θ_H	Σ_T/θ_H	$6(h_1^2 + h_2^2 + h_3^2) + 2(h_1^2 + h_2^2 + h_3^2) + 2h_4^2$	$6(h_1sh_1 + h_2sh_2 + h_3sh_3) + 2(h_1sh_4 + h_2sh_5 + h_3sh_6) + 2h_4sh_7$	$6(h_1 + h_2 + h_3) + 2(h_1 + h_2 + h_3) + 2h_4$
Σ_{SH}	Σ_Z/θ_{SH}	Σ_S/θ_{SH}	Σ_P/θ_{SH}	Σ_T/θ_{SH}	Σ_H/θ_{SH}	$6(sh_1^2 + sh_2^2 + sh_3^2) + 2(sh_4^2 + sh_5^2 + sh_6^2) + 2sh_7^2$	$6(sh_1 + sh_2 + sh_3) + 2(sh_4 + sh_5 + sh_6) + 2sh_7$
Σ_0	Σ_Z/θ_0	Σ_S/θ_0	Σ_P/θ_0	Σ_T/θ_0	Σ_H/θ_0	Σ_{SH}/θ_0	26

Figure 3: Cofactor matrix based on the F-layer from Figure 3.5. This is equivalent to Figure 3.6.