Learning Linear Regression Models using Ring Computation over Factorised Databases

Ruth Wells

Wolfson College

University of Oxford

Supervisor: Professor Dan Olteanu

Dissertation submitted in partial fulfilment of the requirements for the

Master of Science in Computer Science

August 2017

Abstract

This dissertation is concerned with the efficient supervised learning of least-squares linear regression model parameters, specifically where the input training dataset is the result of a natural join query over a relational database. This is typical for a range of scenarios where the data resides in the database and where the aim is to perform analytics, such as regression learning, over such data *in situ*. How the input data is provided to the analytics is crucial: recent work has shown that, using efficient *factorised* representations, learning over query joins can be performed much faster than the materialisation of a *flat* or *listing* representation of the join itself. But the presence of discrete or *categorical* data quantities, and the way in which these are typically *one-hot encoded* in the input data, can result in a non-trivial increase in computational complexity.

We present a new algorithm $\mathbf{F}^{\bigcirc}$, which advances the state of the art in this area of work, to reduce further the computational complexity of learning linear regression models over factorised database joins. This is achieved in two ways: first, the $\mathbf{F}^{\bigcirc}$ algorithm supports the efficient processing of categorical data quantities without requiring them to be one-hot encoded. Second, it employs more powerful computation primitives: state of the art algorithms use semiring operations over the real numbers, that is, addition and multiplication We show, however, that by also employing subtraction and division, the resulting ring computation can achieve asymptotically lower complexity when computing the same database aggregates, especially in the presence of categorical data quantities.

Extending an existing `C++` codebase, we implement the $\mathbf{F}^{\bigcirc}$ algorithm as part of an end-to-end batch gradient descent process, to solve the least squares regression problem for a training dataset comprising an arbitrary natural join query over a relational database. The performance of this implementation is compared against the industry state of the art (specifically an implementation of the **F** algorithm of Schleich, Olteanu and Ciucanu), and against two open-source commercial database systems, **MADlib** and **R**. The implementation of **F** also uses a batch gradient descent approach to solve the least-squares regression problem, but one that decouples the computation of database aggregates from the convergence steps of the batch gradient descent process; crucially, this means that the database aggregates need be calculated only once in the learning process. The ring computation of $\mathbf{F}^{\bigcirc}$ precludes this decoupling, however, meaning that a fresh set of database aggregates must be computed for each convergence step of the batch gradient descent process. **MADlib** and **R** do not use batch gradient descent, instead emplyoing analytical solutions to the least-squares regression problem.

$\mathbf{F}^{\bigcirc}$ was tested using four large-scale real-world retailer datasets of increasing size and numbers of categorical data quantities. Out of the four systems compared, only $\mathbf{F}^{\bigcirc}$ was able to compute model parameters for the larger datasets with greater numbers of categorical variables; on the smaller datasets $\mathbf{F}^{\bigcirc}$ was also 5x faster than the implementation of **F**, in terms of computing a single set of database aggregates. In order to benefit fully from the the theoretical complexity improvements offered by the $\mathbf{F}^{\bigcirc}$ algorithm, however, and to match the performance benefits that **F** gains by calculating database aggregates only once in the end-to-end learning process, further refinements of the practical implementation of $\mathbf{F}^{\bigcirc}$ will be required; we make suggestions for such refinements.

Acknowledgements

I am extremely grateful to Professor Dan Olteanu for agreeing to supervise this project, and for his significant input and guidance over the past few months. I should also like to thank Maximilian Schleich for his help in preparing the practical experiments, and his endless patience in answering my many questions. I have made some good friends during my time in Oxford, especially Kholood, Brittany, and folk at St Andrew's Church – I will miss you! Finally, a big thank you to Ian, Jane and Arianna, and to all my family, for their support and encouragement during the year.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The problem

This project is set in the context of *supervised learning*: the task of inferring a functional relationship within a set of data. For the purposes of this project, the **input** to the task is a *training dataset*, in this case given as a join query Q over a database $\mathbf{D}$, in the form of a set of tuples:

$$\begin{aligned} Q(\mathbf{D}) = \{ & (x_1^{(1)}, x_2^{(1)}, x_3^{(1)}, \ldots, x_n^{(1)}, y_1^{(1)}) \\ & (x_1^{(2)}, x_2^{(2)}, x_3^{(2)}, \ldots, x_n^{(2)}, y_1^{(2)}) \\ & \vdots \\ & (x_1^{(m)}, x_2^{(m)}, x_3^{(m)}, \ldots, x_n^{(m)}, y_1^{(m)})\} \end{aligned}$$

Each tuple $(\mathbf{x}, y) \in Q(\mathbf{D})$ contains a scalar response (regressand) y and a tuple $\mathbf{x}$ of encoding features (regressors).

The **output** from the task is a *linear regression model*, for the purposes of this dissertation a linear relationship that estimates the value of the *label* variable y for any given set of *feature* variables $(x_1, x_2, x_3, \ldots, x_n)$, taking the form:

$$y = h(x_1, x_2, \ldots, x_n) \equiv w_0 + w_1 x_1 + w_2 x_2 + w_3 x_3 + \cdots + w_n x_n = \mathbf{w}^{\mathbf{T}} \mathbf{x} \tag{1.1}$$

(where a constant $x_0 \equiv 1$ is assumed). The **task** itself is thus to determine (*learn*) from the training dataset the values of the *model parameters*, that is, the values of the constants $w_0, w_1, \ldots w_n$.

Linear regression models such as this are of interest because there are many practical contexts where a 'response' variable Y may depend in a linear way on a set of 'controlled' inputs $X_1, X_2, \ldots, X_n$, in that $Y = h(X_1, X_2, \ldots X_n) + \epsilon$ for a deterministic linear (or close to linear) function $h(x_1, x_2, \ldots, x_n)$ and a random *residual error* quantity ϵ (typically assumed to have a normal/Gaussian distribution $\epsilon \sim \mathcal{N}(\mu, \sigma^2)$). Examples of this might include sales of an item given the amounts spent on advertising that product over different media (print, radio, tv), or the blood pressure of a patient given the quantities of their medication and the amount of exercise they take. Alternatively, there may be a correlation between random variables $X_1, X_2, \ldots X_n$, and Y, in that $h(x_1, x_2, \ldots, x_n) = E(Y | X_1 = x_1, X_2 = x_2, \ldots X_n = x_n)$ is linear (or close to linear). Examples of this might include the price of a house, given factors such as local crime statistics, its proximity to local amenities, or the sales of an item given non-controllable factors such as weather. Even if $h(x_1, x_2, \ldots, x_n)$ is itself non-linear, there may be a transformation of variables/data (for example, logarithmic in the case of exponential functions) that reduces h to a linear function, allowing the use of linear regression techniques. More broadly, the term 'linear' regression is a restriction only that h is linear in $\mathbf{w}$, thus ensuring desirable analytical properties with respect to $\mathbf{w}$, but this dissertation considers only the case shown in (1.1).

There is a large body of work on learning regression models (e.g. [6] [14]). Whereas these earlier works assume the input given by a design matrix or relation in general, the work in this dissertation considers that the input is given by the relation representing the result of a join query over a relational database. This is typical for a range of scenarios where the data resides in the database and the aim is to perform analytics, such as regression learning, over such data *in situ* [16][22]. How the input data is provided to the analytics is crucial: recent work has shown that learning over joins can be performed much faster than the materialisation of the join itself![22]. In this dissertation, we push the state of the art in this line of work, by further reducing the computational complexity of learning linear regression models over database joins. This is achieved by employing more powerful computation primitives: the state of the art uses semiring operations over the real numbers, that is addition and multiplication (corresponding naturally to the semiring relational operations of union and Cartesian product). We show, however, that by also employing subtraction and division, the resulting ring computation can achieve asymptotically lower computational complexity. Similar improvements have been seen previously for matrix chain multiplication, for example, Strassen's algorithm, which uses ring computation to decrease the number of operations necessary for matrix multiplication [7].

1.2 Motivation

Here the parameters of the linear regression model are learned by using a standard batch gradient descent technique to minimise a *least-squares objective function*, specifically

$$J(w) = \frac{1}{2m}\sum_{i=1}^{m}\left(y^{(i)} - (w_0 + \sum_{k=1}^{n} w_k x_k^{(i)})\right)^2$$

This requires computation of the gradients of this function with respect to each of the model parameters w_i (one for each query variable X_i and one, w_0 corresponding to the intercept of the function h; in practice, the label y and features x_k are drawn from the columns of the same dataset and, by assuming a fixed value of -1 for its *label parameter*, the label y can be treated symmetrically with the x_k). Each of these gradient computations can be expressed as an aggregate computation over the input dataset. Moreover, since in this case the input dataset is given as a join query over a database, and assuming that each of the variables x_i are continuous (thus able to be aggregated), the gradients can be expressed as aggregates over joins in relational databases, specifically

$$\begin{aligned}
\frac{\partial}{\partial w_0}J(w) &= w_0 + \frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{n} w_k x_k^{(i)} \\
\frac{\partial}{\partial w_j}J(w) &= \frac{w_0}{m}\sum_{i=1}^{m} x_j^{(i)} + \frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{n} w_k x_j^{(i)} x_k^{(i)} \qquad j \in [n] \qquad (1.2)
\end{aligned}$$

This dissertation proposes a novel, most efficient to date, approach to perform these aggregate computations, in particular for the aggregates

$$\mathsf{Grad}_j \triangleq \sum_{i=1}^{m}\sum_{k=1}^{n} w_k x_j^{(i)} x_k^{(i)} \qquad j \in [n] \qquad (1.3)$$

Computation of this set of aggregates poses several practical challenges: one is the efficient computation of the training dataset from the underlying database tables, in terms of the join algorithms used. Another is the size of – and potential redundancy in – the resulting joined data: this can have significant efficiency implications for the computations required over these data.

Example 1.1. *As a preliminary motivating example, consider a training dataset comprising the natural join query*

$$Q_1 = R_1(X_1), R_2(X_2), \cdots, R_n(X_n)$$

where the schema of each relation R_i is a single independent continuous-valued variable X_i, and the size (that is, the number of tuples) of each relation $|R_i| = m$. This query is the special join case of a Cartesian product of n unary relations. In this case, each continuous variable X_i contributes a single parameter w_i to the linear regression model, the size of which is thus n.

A naive approach to calculating the set of n Grad aggregates over Q_1 is first to materialise a *listing representation* (or "flat" representation) of the Cartesian product of the n relations, with size m^n, and then to compute the n^2 quantities

$$\sum_{i=1}^{m^n} x_j^{(i)} x_k^{(i)} \qquad j, k \in [n] \tag{1.4}$$

in a single scan over the join result. Each of these n^2 quantities need be calculated only once over the Cartesian product, taking time $\mathcal{O}(m^n)$; thus the n^2 quantities can be calculated in time $\mathcal{O}(n^2 \cdot m^n)$. Each of the n elements of Grad from equation (1.3) then uses n of the n^2 precomputed quantities, thus taking additional time $\mathcal{O}(n)$ to compute. The resulting overall time complexity to compute the full set of n Grad aggregates is thus

$$\begin{aligned} &\mathcal{O}(n^2 \cdot m^n) + \mathcal{O}(n^2) \\ =&\mathcal{O}(n^2 \cdot (m^n + 1) \\ =&\mathcal{O}(n^2 \cdot m^n) \end{aligned}$$

quadratic in the size n of the linear model (that is, the number of parameters to be learned) and linear in the data size of the join query (the number of its tuples).

State-of-the-art solutions, notably the **F** algorithm of Schleich et al. ([22]), employ a range of techniques to address the size and efficiency challenges noted above. These include: the use of worst-case optimal join algorithms for conjunctive queries, such as the Leapfrog Triejoin ([23]) and the use of succinct *factorised* – rather than *listing* – representations of the join results, to minimise redundancy in the joined data ([18]). For the training dataset $Q_1 = R_1(X_1), R_2(X_2), \cdots, R_n(X_n)$ of Example 1.1, the use of these techniques reduces the overall time complexity to compute the set of n Grad_j aggregates from linear (modulo log factors) in the data size of the *join query* to linear in the size of the *input database*, both defined as the number of their tuples, that is, from $\mathcal{O}(n^2 \cdot m^n)$ to $\mathcal{O}(n^2 \cdot nm) = \mathcal{O}(n^3 m)$. This is clearly an asymptotic improvement over the naive case, but the complexity remains cubic overall in the size of the linear model.

Moreover, the size of the linear model can actually far exceed the number of query variables X_i. *Continuous* query variables X_i (such as weight, distance, etc) each contribute a single parameter to the model. As will be seen, however, query variables that represent *discrete* quantities (that is, *categorical* or 'classification' variables such as, say, colour, shape, or location) are typically *one-hot encoded* within a database in such a way as to require a separate parameter to be learned for each distinct *value* taken by that variable in the query. In the worst case, where all query variables are categorical, the number of model parameters to be learned for the training dataset $Q_1 = R_1(X_1), R_2(X_2), \cdots, R_n(X_n)$ could be as large as the number of values in the datatase, that is $m \cdot n$. In this case, the state of the art **F** computations of Schleich et al. would have complexity $\mathcal{O}((nm)^2 \cdot (nm))$ resulting in an overall complexity that is now cubic in the size of the database, $\mathcal{O}(n^3 m^3)$. This can blow up and become unmanageable for large-scale datasets.

A structural reason for the complexity being quadratic in the size of the model is that the quantities in (1.3) and (1.4) are both expressed purely in terms of the *semiring* operations of multiplication and addition (mirroring the underlying join operations in the query of Cartesian product and union respectively). But the real numbers form a *ring*: they have an additive inverse, that is, subtraction: rewriting calculations to use the power of ring computation can yield asymptotic reductions in complexity. As a very simple example, consider the following rewriting of an expression using ring computation:

$$\sum_{j=1}^{100}\sum_{k=1,j\neq k}^{100}(j\cdot k) \equiv \sum_{j=1}^{100}\Big(j\Big(\sum_{k=1}^{100}k - j\Big)\Big) \tag{1.5}$$

As written, the left-hand expression in this identity, using only semiring operations, involves the sum of 9,900 individual products. But the rewritten right-hand expression involves a sum of 100 k values, which can be computed only once requiring 100 additions. For each of the 100 j values, a single subtraction, mutliplication and addition is then required, thus 400 individual operations in total.

Using a similar technique, it is possible to rewrite the Grad_j quantity of equation (1.3) as

$$\mathsf{Grad}_j \equiv \sum_{i=1}^{m}\Big(w_j x_j^{(i)} x_j^{(i)} + x_j^{(i)}\Big((\sum_{k=1}^{n} w_k x_k^{(i)}) - w_j x_j^{(i)}\Big)\Big)$$

an expression that is now *linear*, rather than quadratic, in the size of the linear regression model.

Building on the state of the art, this project proposes that – with appropriate rewriting of the aggregate calculations to use ring computation – it is possible to achieve further asymptotic complexity improvements on the state-of-the-art, when learning linear model parameters over arbitrary database join queries with both continuous and/or discrete variables.

In the case of query $Q_1 = R_1(X_1), R_2(X_2), \cdots, R_n(X_n)$ of Example 1.1 where all query variables are continuous, using a factorised representation of the query, and rewriting the semiring aggregate calculations to employ ring computation, thus reduces the asymptotic time complexity (modulo log factors) to linear in *both* the size of the database *and* the number of parameters to be learned, that is, from $\mathcal{O}(n^2m^2)$ to $\mathcal{O}(nm)$. Moreover, in the case where all query variables X_i are categorical, the complexity is now reduced to $\mathcal{O}(nm \cdot nm) = \mathcal{O}(n^2m^2)$ compared with $\mathcal{O}(n^3m^3)$ for the state of the art $\mathbf{F}$ algorithm.

1.3 Contributions

The contributions of this dissertation are:

- An algorithm, $\mathbf{F}^{\bigcirc}$, to perform the rewritten computations over factorised representations of arbitrary natural join queries over relational databases. This algorithm builds on the state of the art [22] in two ways: (i) it supports both continuous and categorical query variables, and allows query variables to be excluded completely as model features; and (ii) its computation is expressed in terms of a ring rather than a semiring. This can decrease the algorithmic complexity of the algorithm, especially in the case of one-hot encoded categorical variables (which can lead to a non-trivial increase in complexity).

- Complexity analysis of $\mathbf{F}^{\bigcirc}$, demonstrating that there are classes of natural joins for which it is most efficient to date and, indeed, asymptotically better than state of the art algorithms, such as $\mathbf{F}$.

- Implementation of the $\mathbf{F}^{\bigcirc}$ algorithm, building on and extending an existing `C++` code base for the construction of factorised database representations and linear regression models;

- Experiments to compare the performance of $\mathbf{F}^{\bigcirc}$ against industry state of the art (an implementation of the $\mathbf{F}$ algorithm of Schleich et al.) and against two open-source commercially available relational database systems. Across the systems and datasets tested, only $\mathbf{F}^{\bigcirc}$ was able to compute model parameters for the larger datasets with larger numbers of categorical variables. $\mathbf{F}^{\bigcirc}$ was also 5x faster than the implementation of $\mathbf{F}$ in terms of computing the set of aggregates in (1.2) above.

- Suggestions for further work on the practical implementation of $\mathbf{F}^{\bigcirc}$, in order to benefit fully from the theoretical complexity improvements offered by the algorithm, and to enable it to match its competitors more closely in terms of the end-to-end learning process on smaller datasets (given, for example, the performance benefits that **F** gains by calculating database aggregates only once in the learning process).

1.4 Organisation

This chapter has set out the landscape within which this dissertation is set, that is, the in-database learning of parameters for linear regression models, where the training dataset is the result of a join query over a relational database. By way of a simple example, it has demonstrated that the computational complexity of state of the art solutions is still (at least) quadratic in the size of the linear regression model, and that the presence of discrete-valued data attributes can lead to a further non-trivial increase in complexity, thus motivating the search for a more efficient computational solution. The remainder of this dissertation presents an improved algorithm $\mathbf{F}^{\bigcirc}$, including its implementation in `C++` and experiments to test its performance against the state of the art and against open-source commercially available relational database systems. The structure of the remaining chapters is as follows:

Chapter 2 outlines the key areas of supervised learning and database theory that underpin the technical contribution of this dissertation, and defines the terminology and notation used in subsequent chapters. This includes the specific linear regression model for which a set of parameters needs to be learned, and the typical structure and computational complexity of an algorithm for computing quantities over a *factorised* representation of a database query

Chapter 3 describes the state of the art **F** algorithm of Schleich et al. ([22]) for computing the query aggregates necessary to learn model parameters, including an intuitive analysis of its computational complexity. It then demonstrates that rewriting these aggregate quantities using more powerful computation primitives (specifically ring computation over the real numbers) offers asymptotic complexity improvements over the state of the art, and presents a new algorithm, $\mathbf{F}^{\bigcirc}$, which employs ring computation.

Chapter 4 describes the practical implementation in `C++` of an end-to-end batch gradient descent process involving (i) materialisation of the factorised join query; (ii) implementation of the $\mathbf{F}^{\bigcirc}$ algorithm to compute the required query aggregates; and (iii) embedding the $\mathbf{F}^{\bigcirc}$ computations within a simple batch gradient descent iterative process.

Chapter 5 describes the experiments carried out to compare the performance of the $\mathbf{F}^{\bigcirc}$ implementation against industry state of the art (a development of the **F** system of Schleich et al, namely **DC** [16], and against two open-source commercially available relational database systems, **MADlib** [12] 1.8 (**M**) and **R** [20] 3.0.2.

Chapter 6 describes some current related academic work in the area of machine learning and databases.

Chapter 7 summarises the outcomes of the current project on which this dissertation is based; in light of the experimental outcomes from Chapter 5, it also suggests directions for further work in order to benefit more fully from the theoretical complexity improvements offered by the $\mathbf{F}^{\bigcirc}$ algorithm.

Appendix A comprises the code listing for the new central class of the `C++` implementation of $\mathbf{F}^{\bigcirc}$, namely `CategoricalRegressionOverJoin.cpp`.

Appendix B gives the code of selected new methods that have been added to other classes of the existing `C++` codebase.

Chapter 2

Preliminaries

This chapter outlines the key areas of supervised learning and database theory that underpin the technical contribution of this dissertation, and defines the terminology and notation used. It sets out the specific linear regression model for which a novel approach to learning parameters is presented in Chapter 3, and the additional challenges faced when that model involves discrete-valued *one-hot encoded* quantities. It then describes, with examples, how a database query can be presented in a *factorised* representation; the efficiency gains that this brings in terms of computation over that query; and the typical structure of algorithms to perform computations over factorised representations. It ends with a brief discussion of join algorithms. Chapter 3 then considers specific examples of these algorithms, including the state of the art, and introduces the novel algorithm proposed by this dissertation.

To note that space precludes an exhaustive discussion of the topics below; in each case, further detail can be found in the works referenced within the text.

2.1 Notation and Definitions

$[m, n]$ denotes the set of integers $\{m, m+1, ..., n-1, n\}$. $[n]$ denotes $\{1, 2, ..., n\}$.

2.1.1 Databases.

This project considers relational databases, where a *database* $\mathbf{D}$ is a collection of *relations* R, each over a *schema* $\mathcal{S}$, that is, a set of named *variables* $A, B, C, ...$, where there is a bijection between data attributes and variable symbols. A *tuple* t of a relation R over schema $\mathcal{S}$ is a mapping from $\mathcal{S}$ to a *domain* $\mathcal{D}$ which may comprise both discrete- and continuous-valued quantities. The size of a relation R, denoted $|R|$, is the number of its tuples; the size of a database $\mathbf{D}$, denoted $|\mathbf{D}|$, is the sum of the sizes of its relations. For a database $\mathbf{D}$ with n relations, we thus have $|\mathbf{D}| \leq n \cdot N$ where N is the size of the largest relation in $\mathbf{D}$.

2.1.2 Queries.

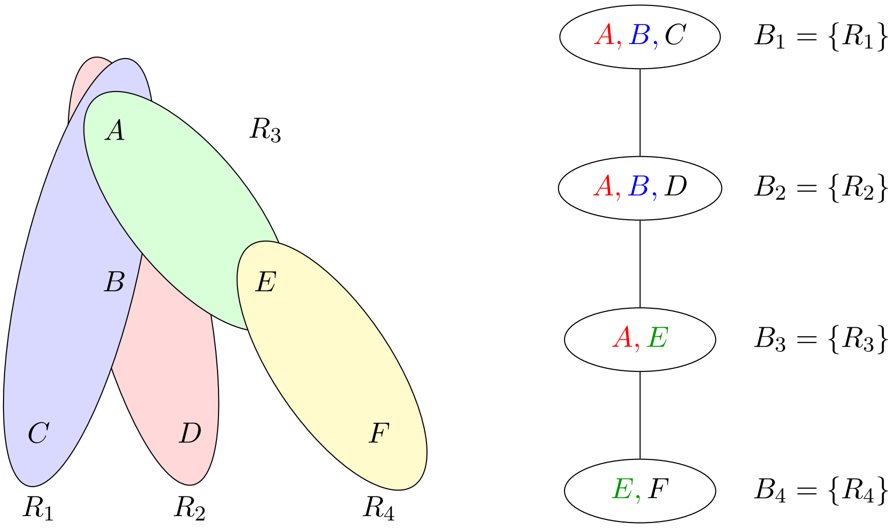
The queries considered are a subset of conjunctive queries of the form

$$Q = \pi_{\mathcal{P}}(\sigma_{\psi}(R_1 \times R_2 \times \cdots \times R_n)) \tag{2.1}$$

specifically *natural join* queries, denoted

$$Q(\mathcal{S}_1 \cup \mathcal{S}_2 \cup \cdots \cup \mathcal{S}_n) = R_1(\mathcal{S}_1), R_2(\mathcal{S}_2), \ldots, R_n(\mathcal{S}_n) \tag{2.2}$$

where the selection condition ψ is that variables with the same name in different queries are equal, and where the projection list $\mathcal{P} = \bigcup_i \mathcal{S}_i$ over all relations R_i. In this dissertation the head variables will be omitted, and the query notation $Q = \ldots$ used. Without loss of generality, it can be assumed that each variable symbol appears at most once in each schema $\mathcal{S}_i$, and that a join of two relations is expressed by having the same variable in the corresponding relation symbols in the query. (In principle the same variable can be used multiple times in a relation



(a) Hypergraph for Q_2

(b) Tree decomposition for Q_2 with fractional width = 1

Figure 2.1: (a) Hypergraph and (b) Tree Decomposition for query $Q_2 = R_1(A,B,C), R_2(A,B,D), R_3(A,E), R_4(E,F)$

symbol, corresponding to selection with an equality condition on corresponding columns of the relation; however, we can assume this trivial operation to have been solved as a filter condition before the join, so that no such selection need be considered afterwards.)

The *size* of a query $|Q|$ is the number of relations n in Q. The *projection size* (or *schema size*) of a query Q is $|\mathcal{P}|$ (in the case where Q is a natural join query, this is simply the number of distinct query variables). The *data size* of a query Q over a database $\mathbf{D}$, $|Q(\mathbf{D})|$, is the number of tuples in $Q(\mathbf{D})$.

Hypergraph of a Query.

Definition 2.1 ([10])**.** *$\mathcal{H} = (\mathcal{V}, \mathcal{E})$ denotes the hypergraph of the query Q, where $\mathcal{V}$ is the set of variables occurring in Q, and $\mathcal{E}$ is the set of hyperedges with one hyperedge per set of variables in a relation symbol R in the body of Q.* □

Example 2.2. *Given the schema $\mathcal{S} = \{A, B, C, D, E, F\}$, relations over $\mathcal{S}$: $R_1(A,B,C)$, $R_2(A,B,D)$, $R_3(A,E)$, and $R_4(E,F)$, and the natural join*

$$Q_2 = R_1(A,B,C), R_2(A,B,D), R_3(A,E), R_4(E,F)$$

a hypergraph $\mathcal{H}(Q_2)$ for Q_2 is shown in Figure 2.1.

Fractional Hypertree Decomposition of a Query Hypergraph.

Definition 2.3 ([10])**.** *Let $\mathcal{H}(\mathcal{V}, \mathcal{E})$ be a hypergraph. A* tree decomposition *of $\mathcal{H}$ is a pair $(T, (B_t)_{t \in \mathcal{V}(T)})$ where*

- *T is a tree (each node of which corresponds to some set of vertices of $\mathcal{H}$), and*
- *$(B_t)_{t \in \mathcal{V}(T)}$ is a family of sets of vertices of $\mathcal{H}$ called bags, such that each hyperedge of $\mathcal{H}$ is contained in some (B_t), and for each vertex v of $\mathcal{H}$ the set $\{t : B_t \ni v\}$ is connected in T.*

A fractional hypertree decomposition *of $\mathcal{H}$ is a triple $(T, (B_t)_{t \in \mathcal{V}(T)}, (\gamma_t)_{t \in \mathcal{V}(T)})$ where $(T, (B_t))$ is a tree decomposition and*

- $(\gamma_t)_{t\in\mathcal{V}(T)}$ *is a family of* weight functions $\mathcal{E}(\mathcal{H}) \mapsto [0,\infty)$ *such that for each* $t \in \mathcal{V}(T)$, γ_t *covers all vertices of* B_t, *i.e.* $\sum_{e\ni v}\gamma_t(e) \geq 1$ *for all* $v \in B_t$.

The weight *of a weight function* (γ_t) *is weight* $(\gamma_t) = \sum_{e\in\mathcal{E}(\mathcal{H})}\gamma_t(e)$, *and the width of the decomposition is* $\max_{t\in\mathcal{V}(T)}\mathrm{weight}(\gamma_t)$. *The* fractional hypertree width *of* $\mathcal{H}$, $\mathrm{fhtw}(\mathcal{H})$, *is the minimum possible width of a fractional hypertree decomposition of* $\mathcal{H}$. □

Example 2.4. *Given the natural join*

$$Q_2 = R_1(A,B,C), R_2(A,B,D), R_3(A,E), R_4(E,F)$$

a tree decomposition of $\mathcal{H}(Q_2)$ *is shown in Figure 2.1(b). Each bag* B_i *comprises the single hyperedge* R_i. *The tree decomposition is connected, since for each query variable A-F the set of nodes of the tree containing that variable form a connected set (highlighted in Figure 2.1(b) in colour). By setting* $\gamma_1(R_1) = \gamma_2(R_2) = \gamma_3(R_3) = \gamma_4(R_4) =$ *1 and* $\gamma_i(R_j) = 0$ *for* $i \neq j$, $\sum_{e\ni v}\gamma_t(e) \geq$ *1 is satisfied for all* $v \in B_t$, *and weight* $(\gamma_t) =$ *1 for all* $t \in \mathcal{V}(T)$. *Thus the width of this hypertree decomposition* $= \max_{t\in\mathcal{V}(T)}\mathrm{weight}(\gamma_t) =$ *1. But* $fhtw(t) \geq$ *1; thus the fractional hypertree width of* $\mathcal{H}(Q) =$ *1.*

Fractional Edge Cover Number of a Query.

Definition 2.5 ([10]). *A* fractional edge cover *of a hypergraph* $\mathcal{H} = (\mathcal{V},\mathcal{E})$ *is a mapping* $\psi : \mathcal{E} \mapsto [0,\infty)$ *such that*

$$\sum_{e\in\mathcal{E},v\in e}\psi(e) \geq 1 \textit{ for all } v \in \mathcal{V}$$

The number $\sum_{e\in\mathcal{E}}\psi(e)$ *is the* weight *of* ψ, *and the* fractional edge cover number, $\rho^*(Q)$ *is the (rational) minimum weight among all fractional edge covers of* $\mathcal{H}(Q)$. □

Informally, $\rho^*(Q)$ is the minimum number of hyperedges of $\mathcal{H}(Q)$ required to cover all vertices of $\mathcal{H}(Q)$ at least once, relaxed to allow fractional cover by individual hyperedges.

Example 2.6. *In the case of* $Q_2 = R_1(A,B,C), R_2(A,B,D), R_3(A,E), R_4(E,F)$, $\rho^*(Q_2) = 3$, *since each of the hyperedges corresponding to* R_1, R_2 *and* R_4 *are necessary to cover the variables C, D and F respectively, but between them they are sufficient to cover all the query variables.*

α-acyclic Queries.

Definition 2.7 ([1]). *An* α*-acyclic query Q is one that satisfies each of five equivalent properties:*

(i) Each instance **I** *of the query that is pairwise consistent is globally consistent.*

(ii) Q has a full reducer.

(iii) Q has a join tree.

(iv) The output of the GYO algorithm on Q is empty.

(v) The hypergraph $\mathcal{H}(Q)$ *is acyclic.* □

Example 2.8. *From Figure 2.1,* $Q_2 = R_1(A,B,C), R_2(A,B,D), R_3(A,E), R_4(E,F)$ *can be seen to have an acyclic hypergraph (informally, there is no set of hyperedges that form a cycle). Thus by (v) in definition 2.7, query* Q_2 *is* α*-acyclic.*

It has also been shown that α-acyclic queries are precisely those that have fractional hypertree width = 1 [9]. Q_2 being α-acyclic is thus consistent with the fractional hypertree width of $\mathcal{H}(Q_2) = 1$ in Example 2.4 and Figure 2.1.

2.1.3 Rings and semirings.

Definition 2.9. *A* ring $(\mathcal{R}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ *is a set $\mathcal{R}$ with closed binary operations $\oplus$ and $\otimes$, the additive identity $\mathbf{0}$, and the multiplicative identity $\mathbf{1}$ satisfying the axioms ($\forall a, b, c \in \mathcal{R}$):*

(i) $a \oplus b = b \oplus a$.

(ii) $(a \oplus b) \oplus c = a \oplus (b \oplus c)$.

(iii) $\mathbf{0} \oplus a = a \oplus \mathbf{0} = a$.

(iv) $\exists - a \in \mathcal{R} : a \oplus (-a) = (-a) \oplus a = \mathbf{0}$.

(v) $(a \otimes b) \otimes c = a \otimes (b \otimes c)$.

(vi) $a \otimes \mathbf{1} = \mathbf{1} * a = a$.

(vii) $a \otimes (b \oplus c) = a \otimes b \oplus a \otimes c$ *and* $(a \oplus b) \otimes c = a \otimes c \oplus b \otimes c$.

A semiring $(\mathcal{R}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ *satisfies all of the above properties except the additive inverse property and adds the axiom that $\mathbf{0} \otimes a = a \otimes \mathbf{0} = \mathbf{0}$. A* commutative *(semi)ring is one for which $a \otimes b = b \otimes a$.* □

Example 2.10. *The number sets $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$ with arithmetic operations $+$ and $\cdot$ and numbers 0 and 1 form commutative rings. The set $\mathcal{M}$ of $(n \times n)$ matrices forms a non-commutative ring $(\mathcal{M}, \cdot, +, 0_{n,n}, I_n)$, where $0_{n,n}$ and I_n are the zero matrix and the identity matrix of size $(n \times n)$. The set $\mathbb{N}$ of natural numbers is a commutative semiring but not a ring because it has no additive inverse. Further examples are the max-product semiring $(\mathbb{R}_+, \max, \times, 0, 1)$, the Boolean semiring $(\{true, false\}, \vee, \wedge, false, true)$, and the set semiring $(2^U, \cup, \cap, \emptyset, U)$ of all possible subsets of a given set U.*

To note that ring computation can be asymptotically more efficient than semiring computation. For example, generalising the example of equation (1.5) to the expression:

$$\sum_{i \in [n]} \sum_{j \in [n], j \neq i} i \cdot j \tag{2.3}$$

Restricting the operations used to semiring addition and multiplication requires the addition of $n(n-1)$ individual products, with complexity $\mathcal{O}(n^2)$. But allowing the ring operation of subtraction (that is, the additive inverse) allows the sum to be re-written as:

$$\sum_{i \in [n]} \left(i \left(\sum_{j \in [n]} j - i \right) \right)$$

With this rewriting, $\sum_{j \in [n]} j$ need be computed only once; this can be done in time $\mathcal{O}(n)$. For each of the n values of i the quantity $i(\sum_{j \in [n]} j - i)$ can then be computed in constant $\mathcal{O}(1)$ time with respect to n (a single subtraction and a single multiplication). The overall complexity of the expression becomes

$$\mathcal{O}(n) + n \cdot \mathcal{O}(1) = \mathcal{O}(n)$$

that is, an asymptotic complexity improvement on the expression in (2.3).

2.2 Least-Squares Linear Regression Model

2.2.1 The Least-Squares Objective Function

The standard linear regression model (for example, [15]) involves a vector of input data variables (or *features*) $\mathbf{x} = x_1 \ldots x_n$, a set of *parameters* of the model $\mathbf{w} = w_0 \ldots w_n$ and an scalar output *response* (or *label*) variable y, between which an underlying linear relationship in $\mathbf{w}$ is assumed (thus ensuring desirable analytical properties with respect to $\mathbf{w}$) [6]:

$$y = h(x_1, x_2, \ldots, x_n) \equiv w_0 + \mathbf{w^T}\phi(\mathbf{x}) \qquad \text{for some function } \phi : \mathbb{R}^n \mapsto \mathbb{R}^n \tag{2.4}$$

For the purposes of this dissertation, we assume the simpler case where h is linear in both $\mathbf{w}$ and $\mathbf{x}$, that is

$$y = w_0 + \sum_{k=1}^{n} w_k x_k$$

The task is to estimate (*learn*) the values of each w_k from a set of *training data*, that is, a set of m observations $x_1^{(i)} \ldots x_n^{(i)}$ for $i = 1 \ldots m$. This can done by finding the values of w_k that minimises a *least-squares objective function* or *loss function* which, for the purposes of this dissertation, is

$$J(\mathbf{w}) = \frac{1}{2m} \sum_{i=1}^{m} \left(y^{(i)} - (w_0 + \sum_{k=1}^{n} w_k x_k^{(i)}) \right)^2 \tag{2.5}$$

2.2.2 Minimising the Least-Squares Objective Function

Minimising the least-squares objective function $J(\mathbf{w})$ is equivalent to finding the value of $\mathbf{w}$ for which the partial derivatives or *gradients* of $J(\mathbf{w})$ with respect to each w_j are zero. Using standard differentiation techniques, this is equivalent to solving the set of linear equations

$$\begin{aligned} \frac{\partial}{\partial w_0} J(\mathbf{w}) &= \frac{-1}{m} \sum_{i=1}^{m} \left(y^{(i)} - (w_0 + \sum_{k=1}^{n} w_k x_k^{(i)}) \right) = 0 \\ \frac{\partial}{\partial w_j} J(\mathbf{w}) &= \frac{-1}{m} \sum_{i=1}^{m} x_j^{(i)} \left(y^{(i)} - (w_0 + \sum_{k=1}^{n} w_k x_k^{(i)}) \right) = 0 \qquad j \in [n] \end{aligned} \tag{2.6}$$

In practice, the label y and features x_k are drawn from the columns of the same dataset; by assuming a fixed value of -1 for its *label parameter*, the label y can be treated symmetrically with the x_k and the partial derivatives of the objective function can be rewritten as

$$\begin{aligned} \frac{\partial}{\partial w_0} J(\mathbf{w}) &= \frac{1}{m} \sum_{i=1}^{m} \left(w_0 + \sum_{k=1}^{n} w_k x_k^{(i)} \right) \\ &= w_0 + \frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{n} w_k x_k^{(i)} = 0 \\ \frac{\partial}{\partial w_j} J(\mathbf{w}) &= \frac{1}{m} \sum_{i=1}^{m} \left(x_j^{(i)} w_0 + \sum_{k=1}^{n} x_j^{(i)} w_k x_k^{(i)} \right) \qquad j \in [n] \\ &= \frac{w_0}{m} \sum_{i=1}^{m} x_j^{(i)} + \frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{n} w_k x_j^{(i)} x_k^{(i)} = 0 \qquad j \in [n] \end{aligned} \tag{2.7}$$

The set of linear equations in (2.7) can be solved by analytical (for example, matrix) techniques: the resulting value of $\mathbf{w}$ is known as the *ordinary least squares* or *ols* solution to the linear regression problem [14]. For large n, however, the analytical approach can be computationally very costly; in such cases, the minimisation can be achieved using an iterative numerical approach involving the partial derivatives of the objective function with respect to each w_j. This dissertation uses one such approach, known as *batch gradient descent* [15].

Specifically, the batch gradient descent algorithm starts with an initial (randomly chosen) value for each w_j and performs repeated updates according to the rule

$$w_j := w_j - \alpha \frac{\partial}{\partial w_j} J(w) \tag{2.8}$$

where α is a step size known as the *learning rate.* The least-squares objective function is convex, thus has a global minimum, and each iteration of (2.8) takes place in the direction of a decreasing gradient. Assuming a suitable (that is, not too large) learning rate α, the batch gradient descent method will always converge to this minimum. A variety of techniques exist for regulating the learning rate, from *adaptive* techniques which adapt the learning rate according to the current values of the gradients, to more sophisticated *backtracking* techniques [8].

The practical challenge is thus how to compute most efficiently the required gradients required in (2.7) from the training dataset. To note that, as currently written, these gradient computations involve the *semiring* operations of addition and multiplication; existing state of the art algorithms for learning linear regression model parameters, such as the **F** algorithm use only these operations to compute the required aggregates [18]. As will be seen below, the novel aspect of this dissertation is that it employs *ring* computation to rewrite the gradients of equation (2.7), resulting in asymptotic efficiency gains when learning the same parameters.

2.2.3 More Sophisticated Linear Regression Techniques

This dissertation focuses only on minimising the least-squares regression function of (2.5). However, if the training data are 'noisy', this approach can result in overfitting, in the sense that the computed parameters for the model are overly sensitive to the noise in the training data rather than to the underlying shape of the training data. To avoid this, the least squares regression model can be extended to accommodate *penalty terms*, such as the ℓ_2-norm used in Ridge Linear Regression; this results in a smoother fit between the training data and the calculated model [14].

2.3 One-Hot Encoding of Categorical Variables

Equations (2.5) to (2.7) are valid in the case where the model features are continuous-valued quantities, such as weight or distance; in this case, each continuous variable maps to a single scalar x_i for some i. There are many real-life situations, however, where discrete-valued *categorical* (or 'classification') features may determine the value of a label: for example, where the price of a dress is dependent on its colour, or where the monthly sales figures for a food store are dependent on the chain that the store is part of. In fact, categorical features may constitute the majority of features in commercial machine learning applications [16]. This poses an immediate practical challenge for the linear regression model: that of how to carry out the batch gradient descent computations on discrete features. What, for example, does $w_k x_k$ mean when x_k represents a colour or a brand? One possible approach is to retain a single feature for the variable, but to encode each possible category numerically, for example in the case of dress colours, red = 0, green = 1, blue = 2. But any particular choice of encoding values then imposes an implicit ordering/skew on the model parameter w_k for this feature; put another way, a *single* model parameter cannot capture impartially a *multi*-way choice.

A standard solution in machine learning is to use *one-hot encoding* whereby each single categorical data variable is encoded using an indicator vector of features, with one model parameter per value in the active domain of the feature [11]. Thus, in the case of dress colours, where the active domain of the variable colour comprises, say, $\{\mathsf{red}, \mathsf{green}, \mathsf{blue}\}$, the categorical variable colour is now mapped to an indicator vector $\mathbf{x}_{\mathsf{colour}}$, where $\mathbf{x}_{\mathsf{colour}} = \{x_{\mathsf{red}}, x_{\mathsf{green}}, x_{\mathsf{blue}}\}$. The value red for colour is then encoded as [1, 0, 0], green as [0, 1, 0] and blue as [0, 0, 1], with associated model parameters w_{red}, w_{green} and w_{blue}. (The name one-hot encoding derives from the fact that only one bit of each indicator vector can be 1 or 'hot'.) In the sense that the

R_1

A	B	C
a_1	b_1	c_1
a_1	b_1	c_2
a_1	b_2	c_1
a_2	b_1	c_2

R_2

A	B	D
a_1	b_1	d_1
a_1	b_1	d_2
a_1	b_2	d_1
a_2	b_1	d_2

R_3

A	E
a_1	e_1
a_1	e_2
a_2	e_1
a_2	e_3

R_4

E	F
e_1	f_1
e_1	f_2
e_2	f_1
e_3	f_2

(a) All query variables continuous

R_1

A	B	$C : c1$	$C : c2$
a_1	b_1	1	0
a_1	b_1	0	1
a_1	b_2	1	0
a_2	b_1	0	1

R_2

A	B	D
a_1	b_1	d_1
a_1	b_1	d_2
a_1	b_2	d_1
a_2	b_1	d_2

R_3

A	E
a_1	e_1
a_1	e_2
a_2	e_1
a_2	e_3

R_4

E	$F : f1$	$F : f2$
e_1	1	0
e_1	0	1
e_2	1	0
e_3	0	1

(b) Query variables C and F categorical

Figure 2.2: Example data for relations R_1 to R_4, (a) where all query variables represent continuous data quantities; and (b) the same data, one-hot encoded, when query variables C and F represent categorical (discrete) data quantities.

vectors (1, 0, 0), (0, 1, 0) and (0, 0, 1) are ‘equidistant’, one-hot encoding does not impose any ordering/skew on the model parameters in the batch gradient descent computations. Moreover, the computations themselves apply exactly as for continuous features (notwithstanding the fact that any values x_k corresponding to a single one-hot encoded feature will take only the value 0 or 1).

Example 2.11. *Consider again the relations $R_1(A, B, C)$, $R_2(A, B, D)$, $R_3(A, E)$, and $R_4(E, F)$. Figure 2.2(a) shows a sample listing representation of the relations in the case where all query variables represent continuous data quantities. Figure 2.2(b) shows a one-hot encoded representation of the same relations in the case where query variables C and F are categorical query variables representing discrete data quantities.*

A significant drawback of one-hot encoding, however is that the *size* of the linear regression model – that is, the number of its model parameters – is now determined by the number of possible values for the categorical variables within the training dataset, and can be of the same order of magnitude as the size of the underlying database. For example, suppose that dress colour appeared as variable of a single database relation R_{max} – the largest relation in the database – and that every dress was a different colour. The number of parameters for dress colour would thus be the same as the number of tuples in that relation (recall that $|\mathbf{D}| = \mathcal{O}(|R_{max}|)$).

2.4 Factorised Databases

2.4.1 Listing Representations

The traditional way of storing relational data is in ‘flat’ tabular format, i.e. explicitly enumerating each individual tuple in each relation. Materializing the join of two or more relations - for example, a natural join on common variables – in such a *listing representation* can lead to a high degree of data redundancy/duplication.

As an example consider again the relations $R_1(A, B, C)$, $R_2(A, B, D)$, $R_3(A, E)$, and $R_4(E, F)$ and their natural join $Q_2 = R_1(A, B, C), R_2(A, B, D), R_3(A, E), R_4(E, F)$ where the tuples in each relation are as shown in Figure 2.2(a).

There is clearly multiple data redundancy in the listing representation, highlighted in Figure 2.3. If quantities such as aggregates are required to be computed over the join, this

$$Q_2 = R_1(A,B,C), R_2(A,B,D), R_3(A,E), R_4(E,F)$$

A	B	C	D	E	F
a_1	b_1	c_1	d_1	e_1	f_1
a_1	b_1	c_1	d_1	e_1	f_2
a_1	b_1	c_1	d_1	e_2	f_1
a_1	b_1	c_1	d_2	e_1	f_1
a_1	b_1	c_1	d_2	e_1	f_2
a_1	b_1	c_1	d_2	e_2	f_1
a_1	b_1	c_2	d_1	e_1	f_1
a_1	b_1	c_2	d_1	e_1	f_2
a_1	b_1	c_2	d_1	e_2	f_1
a_1	b_1	c_2	d_2	e_1	f_1
a_1	b_1	c_2	d_2	e_1	f_2
a_1	b_1	c_2	d_2	e_2	f_1
a_1	b_2	c_1	d_1	e_1	f_1
a_1	b_2	c_1	d_1	e_1	f_2
a_1	b_2	c_1	d_1	e_2	f_1
a_2	b_1	c_2	d_2	e_1	f_1
a_2	b_1	c_2	d_2	e_1	f_2
a_2	b_1	c_2	d_2	e_3	f_2

Figure 2.3: Listing representation of $Q_2 = R_1(A,B,C), R_2(A,B,D), R_3(A,E), R_4(E,F)$ where the tuples in each relation are as shown in Figure 2.2(a).

redundancy will lead to corresponding computational inefficiencies, where effectively identical sub-computations are performed multiple times over the duplicated data. For example, to compute the aggregate SUM($A \times E \times F$) over the materialized listing representation above would result in a sum of 18 individual products, involving multiple computations of each of $a_1 \times e_1 \times f_1$, $a_1 \times e_1 \times f_2$, and $a_1 \times e_2 \times f_1$.

2.4.2 Size Bounds for Listing Representations

Proposition 2.12. *Given a join query Q*

- *for any database $\mathbf{D}$, $Q(\mathbf{D})$ has a listing representation with size*

$$\mathcal{O}(|\mathbf{D}|^{\rho^*(Q)}) \tag{2.9}$$

  where $\rho^*(Q)$ is the fractional edge cover number of $\mathcal{H}(Q)$ [10]*;*

- *there exist arbitrarily large databases $\mathbf{D}$ for which $Q(\mathbf{D})$ has a listing representation with size*

$$\mathbf{\Omega}\left((|\mathbf{D}|/|\mathcal{P}|)^{\rho^*(Q)}\right),$$

  where $|\mathcal{P}|$ is the projection (schema) size of Q [3]*.* □

Example 2.13. *$Q_2 = R_1(A,B,C), R_2(A,B,D), R_3(A,E), R_4(E,F)$ – for which, as seen above, $\rho^*(Q_2) = 3$ – admits a listing representation of size $\mathcal{O}(|\mathbf{D}|^3)$.*

Example 2.14. *A hypergraph $\mathcal{H}(Q_1)$ for the query $Q_1 = R_1(X_1), R_2(X_2), \cdots, R_n(X_n)$ is shown in Figure 2.4. Clearly, all n hyperedges are required to cover the n nodes of $\mathcal{H}(Q_1)$, thus $\rho^*(Q_1) = n$. A theoretical bound for the data size of the listing representation of Q_1 is thus $\mathcal{O}(|\mathbf{D}|^n)$; this is consistent with the actual size of the Cartesian product m^n if $|R_i| = m$.*

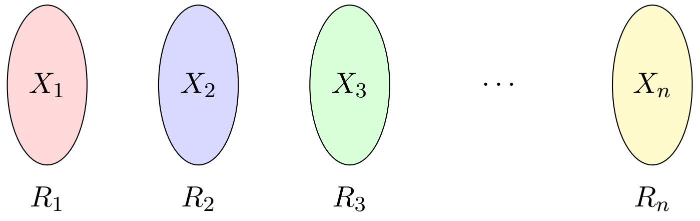


Figure 2.4: Hypergraph for query $Q_1 = R_1(X_1), R_2(X_2), \cdots, R_n(X_n)$

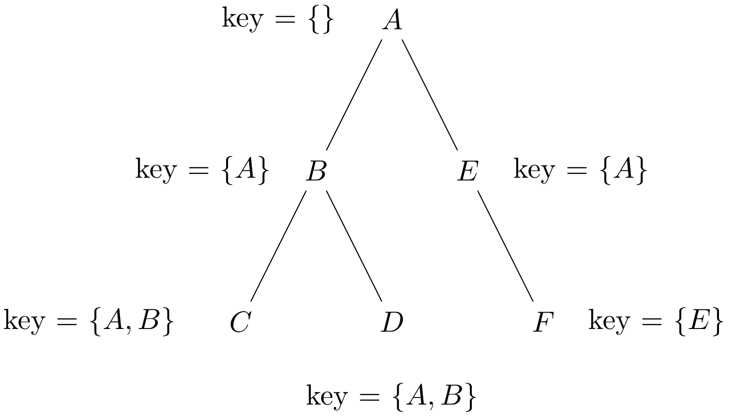


Figure 2.5: d-tree for natural join $Q_2 = R_1(A, B, C), R_2(A, B, D), R_3(A, E), R_4(E, F)$

2.4.3 Factorised Representations

Factorised representations (or *factorisations*) of databases provide a succinct (also lossless and complete) way of representing relational data as relational algebra expressions, using unions, products and (singleton) relations [19]. This is done in a way that exploits the underlying structure and interdependencies of the relations within a query (irrespective of the size and actual content of those relations) so as to avoid the explicit enumeration of duplicate values. This underlying query structure is encapsulated in a *d-tree*, defined as follows:

Definition 2.15 ([19])**.** *An f-tree over a schema $\mathcal{S}$ is a rooted forest with each node labelled by a non-empty subset of $\mathcal{S}$ such that each attribute of $\mathcal{S}$ occurs exactly one node. A* d-tree *is an f-tree in which each node $\mathcal{A}$ is annotated by a set of attributes key($\mathcal{A}$) such that*
— key($\mathcal{A}$) $\subseteq$ anc($\mathcal{A}$);
— key($\mathcal{A}$) is a union of nodes; and
— for any child $\mathcal{B}$ of $\mathcal{A}$, key($\mathcal{B}$) $\subseteq$ key($\mathcal{A}$) $\cup\, \mathcal{A}$;
where the set key($\mathcal{A}$) specifies the ancestor attributes of $\mathcal{A}$ on which the attributes in the subtree rooted at $\mathcal{A}$ may depend.

Example 2.16. *A d-tree for $Q_2 = R_1(A, B, C), R_2(A, B, D), R_3(A, E), R_4(E, F)$ is shown in Figure 2.5.*

A d-tree can be constructed from static analysis of a query over a given database schema (that is, without regard to the actual tuples of the database) Some points to note:

Dependent variables – for example, variables within a single relation – lie on a single root-to-leaf path.Thus, in the case of Q_2, A, B, C lie on a single path, as do A, B, D, A, E, and E, F. The converse is not necessarily true, however, as a variable may depend on only a strict subset of its ancestors. For example, variable F of Q_2 has $anc(F) = \{A, E\}$ but $key(F) = \{E\}$. This is evident in the listing representation of Figure 2.3: for example, if $E = e_1$ than F can always take the values f_1 or f_2, irrespective of the value of A; similarly, if $E = e_2$ then F can only take the value f_1, irrespective of the value of A.

Branching within a d-tree indicates conditional independence between query variables, conditioned on the values of their common ancestors/keys within the tree. For example, the branching to B and E below A indicates that B and E are independent conditioned on the value of A; the branching to C and D below B indicates that C and D are independent conditioned on the values for A and B. It is precisely this conditional independence that leads to redundancy in a listing representation of the query, as it results in the Cartesian product of some subset of data values. This is also evidenced in Figure 2.3, where, for any given value of A, the listing representation must include every possible combination of B and E values and, for any given A, B pair, every possible combination of C and D values.

The *factorised representation* (or *factorisation*) of a query employs a nested tree structure which mirrors the branching of the query's d-tree. This enables conditionally independent variable values from separate relations, which would give rise to a Cartesian product in a listing representation of the join, to be stored separately and – crucially – only once, rather than multiple times. Factorised representations are relational algebra expressions, comprising unions, Cartesian products and *singleton* relations involving a single variable and value, $\langle A : a\rangle$. This dissertation considers specifically *factorised representations with definitions*, defined formally as follows:

Definition 2.17 ([19]). *A* factorised representation with definitions *is a set of named expressions $\{N_1 : D_1, \ldots, N_n := D_n\}$, where each name N_i is a unique symbol and each D_i is an expression with products, unions, singletons and names of other expressions. Formally, an expression over a schema $\mathcal{S}$ is one of the following:*

- *$\emptyset$, representing the empty relation over $\mathcal{S}$,*
- *$\langle\rangle$, representing the relation consisting of the nullary tuple, if $\mathcal{S} = \emptyset$,*
- *$\langle A : a\rangle$, representing the unary relation with a single tuple with value a, if $\mathcal{S} = \{A\}$ and a is a value in the domain $\mathcal{D}$,*
- *$(E_1 \cup E_2 \cdots \cup E_n)$ representing the union of the relations represented by E_i, where each E_i is an expression over $\mathcal{S}$,*
- *$(E_1 \times E_2 \cdots \times E_n)$ representing the Cartesian product of the relations represented by E_i, where each E_i is an expression over schema $\mathcal{S}_i$, such that $\mathcal{S}$ is the disjoint union of all $\mathcal{S}_i$,*
- *a name N_i of another expression D_i over $\mathcal{S}$, representing the same relation as D_i.* □

Example 2.18. *The natural join query $Q_2 = R_1(A, B, C), R_2(A, B, D), R_3(A, E), R_4(E, F)$, where the tuples in each relation are as shown in Figure 2.2(a), has a factorisation as shown in Figure 2.6.*

The factorisation in Figure 2.6 has eliminated much of the redundancy in the listing representation of the query: for example, the sub-tuple a_1, b_1, c_1 is now recorded only once, compared with six times in the listing representation of Figure 2.3. But there is still some residual redundancy in Figure 2.6: specifically that the two sub-trees rooted at e_1 are identical. Recall that $key(F) = \{E\} \subset anc(F) = \{A, E\}$, because F is independent, conditioned on E, from variable A. Thus any value of E gives rise to the same set of values of F in the query result, irrespective of where in the query representation that value of E occurs. This enables further streamlining of the factorisation by only recording the values of F for a given value of E against the *first* occurrence of that value of E. If that value of E occurs multiple times in the query result, subsequent occurrences can simply reference the first, a process referred to as *caching*. Figure 2.7 shows the factorisation of Q_2 with caching.

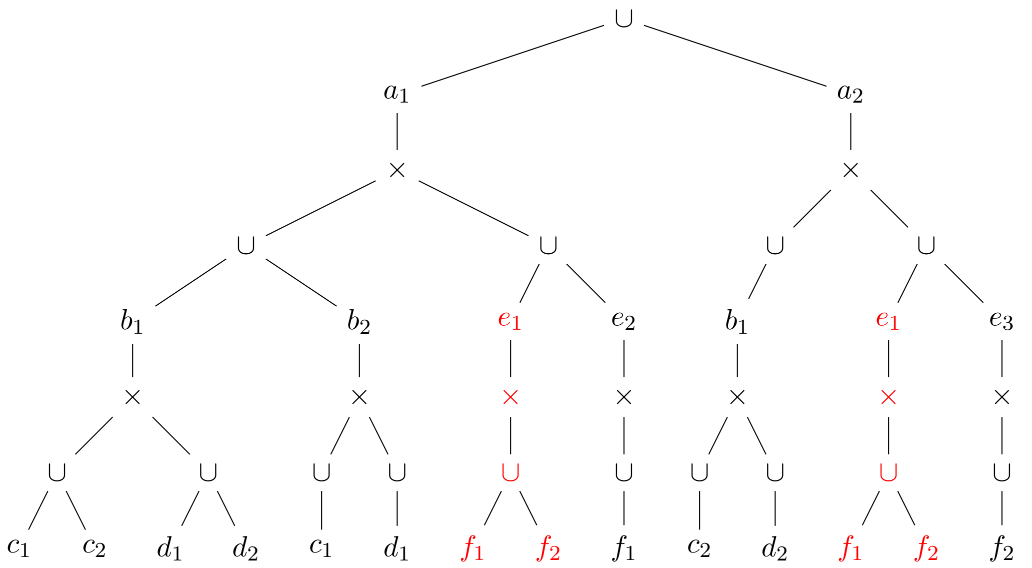


Figure 2.6: Factorisation of $Q_2 = R_1(A,B,C), R_2(A,B,D), R_3(A,E), R_4(E,F)$ where the tuples in each relation are as shown in Figure 2.2(a), and with residual redundancy highlighted.

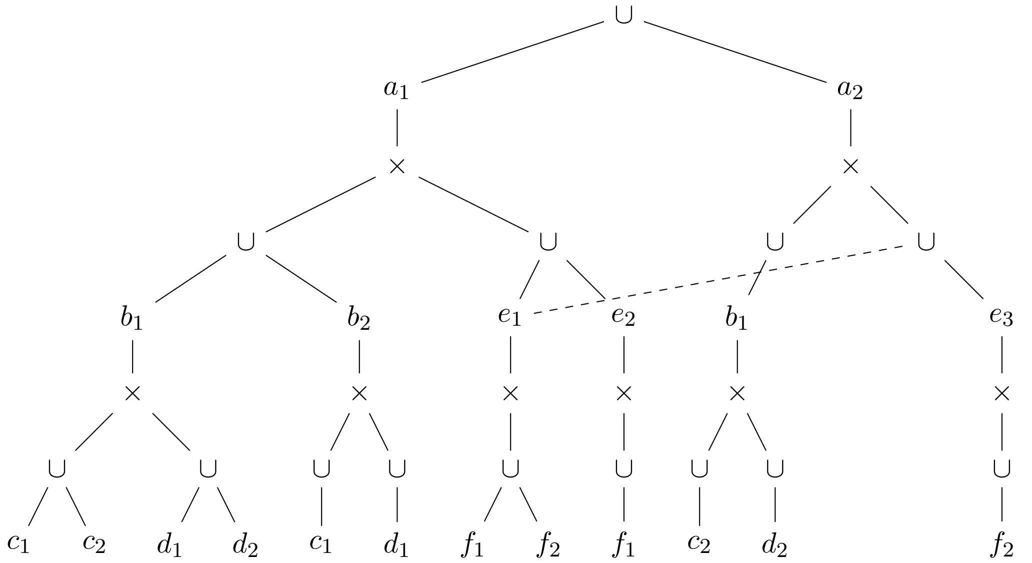


Figure 2.7: Factorisation of $Q_2 = R_1(A,B,C), R_2(A,B,D), R_3(A,E), R_4(E,F)$, where the tuples in each relation are as shown in Figure 2.2(a), and using caching to remove residual redundancy.

2.4.4 Size Bounds for Factorisations

For this simple example, the *size* of the factorisation of Q_2 – that is, the total number of symbols a_1, c_2, etc, $\cup$, $\times$, used in the representation – is only 47 (42 with caching), compared with $18 \times 6 = 108$ (that is, the data size of the query $\times$ the size of the query schema) in the listing representation of Q_2. More generally, a theoretical upper bound for the size of the factorized representation of an arbitrary equi-join query Q over database $\mathbf{D}$ (analogous to equation 2.9 for listing representations) exists:

Proposition 2.19 ([19]). *Given a join query Q*

(i) for any database $\mathbf{D}$, $Q(\mathbf{D})$ has a factorised representation with size

$$\mathcal{O}(|\mathcal{P}|^2 \cdot |\mathbf{D}|^{fhtw(Q)}) \tag{2.10}$$

where $\mathcal{P}$ is the number of query variables, and $fhtw(Q)$ is the fractional hypertree width *of the query Q, satisfying $1 \leq fhtw(Q) \leq \rho^*(Q) \leq |Q| =$ number of relations in Q;*

(ii) There exist arbitrarily large queries Q for which $fhtw(Q) = 1$ and $\rho^*(Q) = |Q|$, where the gap between $fhtw(Q)$ and $\rho^*(Q)$ can be as great as $|Q|$. □

Two points to note from Proposition 2.19: first, for any given natural join query Q, the number of query variables is fixed and $Q(\mathbf{D})$ has a factorised representation with size $\mathcal{O}(|\mathbf{D}|^{fhtw(Q)})$. This simplification makes for more straightforward comparisons of the performance of different algorithms over the same factorised representation in Chapter 3. And, second, in case (ii), from (2.10) $Q(\mathbf{D})$ admits a factorised representation with size

$$\begin{aligned} &\mathcal{O}(|\mathcal{P}|^2 \cdot |\mathbf{D}|^{fhtw(Q)}) \\ =&\mathcal{O}(|\mathcal{P}|^2 \cdot |\mathbf{D}|^1) \\ =&\mathcal{O}(|\mathcal{P}|^2 \cdot |\mathbf{D}|) \end{aligned}$$

where $\mathcal{P}$ is the (fixed) number of (distinct) query variables; and from (2.9), $Q(\mathbf{D})$ admits a listing representation with size

$$\begin{aligned} &\mathcal{O}(|\mathbf{D}|^{\rho^*(Q)}) \\ =&\mathcal{O}(|\mathbf{D}|^{|Q|}) \\ =&\mathcal{O}(|\mathbf{D}|^n) \end{aligned}$$

where n is the number of relations in Q. The size of the factorised representation is therefore exponentially smaller than that of the corresponding listing representation.

Example 2.20. *The query $Q_1 = R_1(X_1), R_2(X_2), \cdots, R_n(X_n)$ has the hypergraph shown in Figure 2.4, d-tree shown in Figure 2.8 and factorisation shown in Figure 2.9 (in the case where $|R_1| = |R_2| = \cdots = |R_n| = m$). It can be seen from the hypergraph that Q_1 is α-acyclic; thus $fhtw(Q) = 1$. $\mathcal{P}$ = number of query variables = n; a theoretical upper bound on the size of the factorisation is therefore $\mathcal{O}(|\mathcal{P}|^2 \cdot |\mathbf{D}|) = \mathcal{O}(n^2 \cdot nm) = \mathcal{O}(n^3m)$; this is consistent with the actual size of the factorisation = $(mn + n + 1)$ in Figure 2.9, and exponentially smaller than the data size of the Cartesian product = m^n.*

Intuitively, the benefits of factorisation (in terms of size and also, therefore, the reduction in the time required to perform any computations over the factorisation) increase as the level of conditional independence between the variables of the underlying relations increases, that is, the greater the degree of branching in the d-tree/factorisation.

As well as deriving size bounds for the factorisation of an equi-join query Q over database $\mathbf{D}$, previous work has presented a worst-case optimal recursive algorithm to compute such a representation [19]:

X_1
$key = \{\}$
X_2
$key = \{\}$
$\cdots$
X_n
$key = \{\}$

Figure 2.8: D-tree for natural join $Q_1 = R_1(X_1), R_2(X_2), \cdots, R_n(X_n)$

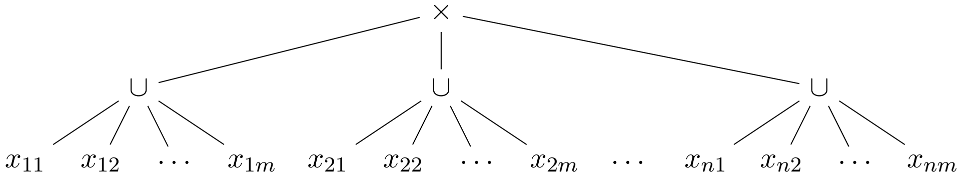

$\times$
$\cup$
$\cup$
$\cup$
x_{11} x_{12} $\cdots$ x_{1m} x_{21} x_{22} $\cdots$ x_{2m} $\cdots$ x_{n1} x_{n2} $\cdots$ x_{nm}

Figure 2.9: Factorisation of $Q_1 = R_1(X_1), R_2(X_2), \cdots, R_n(X_n)$ where $|R_i| = m$

Proposition 2.21 ([19]). *Given any conjunctive query Q and a database $\mathbf{D}$, a factorised representation of $Q(\mathbf{D})$ can be computed in time*

$$\mathcal{O}(|\mathbf{D}|^{fhtw(Q)} \cdot log|\mathbf{D}|)$$

with respect to data complexity. □

Crucially, the factorisation can be obtained directly from the underlying database and the d-tree for the query; there is no need to materialise a listing representation of the join as an intermediate step. In the case of α-acyclic queries (for which $fhtw(Q) = 1$) this complexity becomes $\mathcal{O}(|\mathbf{D}| \cdot log|\mathbf{D}|)$.

2.4.5 Computation over Factorised Datasets

The tree structure of a factorisation lends itself naturally to recursive computations over a traversal of the tree, replacing the semiring operations of Union ($\cup$) and Cartesian product ($\times$) with appropriate arithmetic semiring (or ring) operations. For example, to compute the sum of all values in a factorised dataset, union can simply be replaced by addition, and Cartesian Product by multiplication. The structure of a 'typical' algorithm for computing some *quantity* over a factorisation E is given in Figure 2.10. Some highlighted points to note:

This algorithm uses caching (*1, 4*). Recall, for example, that variable F in query Q_2 has $key(F) = \{E\} \neq anc(F) = \{A, E\}$ – the values of F in any given tuple of the query depend only on the value of its key E. An algorithm looking to perform a computation on the set of F values for a particular value of E therefore first checks to see if that E-value has been seen previously: if so, the results of the previous computation for F given that value of E can be retrieved from the cache and used (*1*); if not, the computation can be performed, and the results stored in the cache (*4*) for the next time the same key is encountered.

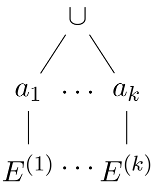
The purpose of the *count* (*2, 5*) and *multiplier* (*6*) quantities is to ensure that the computation takes account of every tuple in the listing representation of the join, effectively 'decompressing' the factorisation, but doing so by scaling (once) each computation 'g', rather than carrying out the same computation 'g' multiple times (a process that can be thought of as pushing the computation 'g' 'through' the join).

The overall complexity of this algorithm depends on the complexity of function 'f' – executed once per value symbol in the factorisation (*3*) – and the complexity of function 'g' – executed once per variable symbol in the factorisation *per* branch below that symbol in the factorisation (*7*) – that is, per branch below that variable in the d-tree). This gives an intuitive computational complexity for typical-algo of

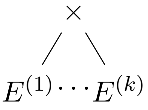
$$\mathcal{O}(\text{size of factorisation}(\mathcal{O}(f) + b \cdot \mathcal{O}(g))) \tag{2.11}$$

where b is some measure of the 'average degree of branching' within the d-tree. (As will be seen in the following chapter, in the case of the $\mathbf{F}$ and $\mathbf{F}^{\bigcirc}$ algorithms to learn linear regression model

```
typical-algo (Factorization E, varMap)
switch $E$:
```



```
$A = var(E)$;    keyMap = $\pi_{key(A)}$(varMap);                                Notes
if ($key(A) \neq anc(A)$) {
    $(count, quantity) = cache$[keyMap]; if ($count \neq 0$) break;               1
}
$count = 0$;
if ($\forall j \in [k] : E^{(j)} = \emptyset$) then foreach $j \in [k]$ do $count^{(j)} = 1$;
else foreach $j \in [k]$ do {
    $(count^{(j)}, quantity^{(j)})$ = typical-algo($E^{(j)}$, varMap × $\{(A : a_j)\}$);
    $count$ += $count^{(j)}$;                                                    2
    $quantity = f(count^{(j)}, quantity^{(j)})$;   // some function $f$           3
}
if ($key(A) \neq anc(A)$)   cache[keyMap] = $(count, quantity)$;                 4
```



```
$count = 1$;
foreach $j \in [k]$ do {
    $(count^{(j)}, quantity^{(j)})$ = typical-algo($E^{(j)}$, varMap);
    $count *= count^{(j)}$;                                                      5
}
foreach $j \in [k]$ do {
    $multiplier^{(j)} = count/count^{(j)}$;                                      6
    $quantity = g(count^{(j)}, quantity^{(j)})$;  // some function $g$           7
}
return ($count, quantity$);
```

Figure 2.10: Structure of a typical algorithm for performing recursive computation to calculate some *quantity* over a factorized join E.

parameters, the complexity of 'f' and 'g' may also depend on the depth of their respective union and Cartesian product operators within the factorisation tree).

Two final points to note: first, recall that the size of a factorisation relative to the corresponding listing representation of a join *decreases* with the degree of branching in the d-tree. As can be seen from equation (2.11) the complexity of a computational algorithm over a factorisation of a given size *increases* with the degree of branching. Overall, however, the reduction in the size of the factorisation can far outweigh the increased complexity of the algorithm from the branching within the d-tree. In fact, even taking into account the time to construct the factorisation of the query, the overall time to perform computation over a factorisation can still be log-linear in the size of the database, compared with potentially exponential for the same computation over a flat representation of the query. Second, equation (2.11) suggests that efficiency gains in the 'branching computation' 'g' may have greater relative benefits for the overall complexity of the algorithm than the same efficiency gains in the 'union computation' 'f'. Function 'g' would therefore be a particularly good candidate for rewriting to replace semi-ring operations with ring operations. As will be seen in the following chapter, the $\mathbf{F}^{\bigcirc}$ algorithm developed in this dissertation achieves precisely this improvement over current state-of-the-art algorithms, such as **F**, for learning linear regression model parameters.

2.5 Join Algorithms: Leapfrog Triejoin

Typical database join algorithms (such as sort-merge join and block-nested loop join) are *relation-oriented*, that is, individual relations within the query are joined in turn. Where a query involves more than two relations, the efficiency of the overall join process is highly dependent on the *query plan* determining the order in which the individual relations are to be added to the join. The Leapfrog Triejoin, on the other hand, is a *variable-oriented* join algorithm for constructing listing (flat) representations of *full conjunctive queries*: queries taking the form of equation 2.1 with the restriction that the projection list $\mathcal{P} = \bigcup_i \mathcal{S}_i$ over all relations R_i (thus a superset of the natural join queries considered in this project) [23], [24]. A Leapfrog Triejoin for query Q with schema $\mathcal{S}(X_1, X_2, \dots X_n)$ is constructed using a backtracking search process on individual variables of the query schema. First, each of the relations involved in the query are joined on variable X_1; for each suitable value of X_1 (if any) each of the relations are then joined on variable X_2, then X_3, and so on, until a tuple $(x_1, x_2, \dots x_n)$ has been created that satisfies the query condition ψ. The search then *backtracks* to ensure that, for each value x_1, every possible combination of x_2, is enumerated and, for each (x_1, x_2) every value of x_3 is enumerated, and so on. Leapfrog Triejoin is intuitively simple to understand and straightforward to implement recursively. It has also been proved to be *worst-case optimal* (that is, no other algorithm has a better computational complexity for handling worst-case queries), computing a listing representation of query Q in time $\mathcal{O}(|\mathbf{D}|^{\rho^*(Q)})$ modulo log factors. Moreover, there are classes of queries for which Leapfrog Triejoin strictly out-performs other more sophisticated worst-case join algorithms [24].

The basic building block of the Leapfrog Triejoin is the *leapfrog join*, a variant of sort-merge join which simultaneously joins a set of unary relations over the same query variable (with computational complexity proportional to the size of the smallest input relation) [23]. By choosing a variable order consistent with that of the d-tree for a query Q, the same leapfrog join technique can used to construct a factorised representation of Q. For example, in the case of query $Q_2 = R_1(A, B, C), R_2(A, B, D), R_3(A, E), R_4(E, F)$, leapfrog join can first be used to join the relations on A (that is, R_1, R_2, R_3). For each possible join value $\langle A : a \rangle$, leapfrog join can then be used to join the relations on B (that is, R_1, R_2) and E (that is, R_3, R_4), and so on. The resulting tree structure of join values corresponds precisely to the factorised representation of Q as depicted in Figure 2.6; the leapfrog join approach can also incorporate caching, resulting in the representation in Figure 2.7.

Chapter 3

$\mathbf{F}^{\bigcirc}$: Ring-based Gradient Computation

Chapter 2 described the linear regression model for which a set of parameters needs to be learned, and the typical structure of an algorithm for computing quantities over the *factorised* representation of a database query. This chapter brings these two together: it describes a state of the art algorithm, **F** ([22]) for computing the query aggregates necessary to learn those parameters (where the query is the training dataset for the linear regression model) and presents an intuitive analysis of its computational complexity. It then demonstrates that, with a rewriting of these aggregate quantities using more powerful computation primitives (specifically ring computation over the real numbers), an improved algorithm, $\mathbf{F}^{\bigcirc}$, exists which offers asymptotic complexity improvements over the state of the art. Chapter 4 then describes the practical implementation in `C++` of the $\mathbf{F}^{\bigcirc}$ algorithm, as part of an end-to-end batch gradient descent supervised learning process.

3.1 F: State of the Art Gradient Computation

The state of the art for learning the parameters of linear regression models over factorised joins is the work of Schleich et al., specifically the **F** system. This includes algorithms both for computing the required aggregates over a materialised factorisation of the result of any join query, and for simultaneous computation of the factorisation and the aggregates in memory [22]. For this dissertation, the focus is the computation of aggregates over a materialised factorisation.

Recall from (2.7) that the gradients required for a single iteration of the batch gradient descent algorithm are

$$\begin{aligned}
\frac{\partial}{\partial w_0} J(w) &= w_0 + \frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{n} w_k x_k^{(i)} \\
\frac{\partial}{\partial w_j} J(w) &= \frac{w_0}{m} \sum_{i=1}^{m} x_j^{(i)} + \frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{n} w_k x_j^{(i)} x_k^{(i)} \qquad j \in [n]
\end{aligned} \tag{3.1}$$

where $w_0 \dots w_n$ are the current values of the model parameters. To note that m is now the size of the materialised factorised representation of the query over which the aggregates are being computed: from Proposition 2.19 this size is $\mathcal{O}(|\mathbf{D}|^{fhtw(Q)})$. The terms in these gradients correspond to aggregates over the training dataset: in the case of the **F** algorithm [22] these aggregates were defined as follows:

$$C \triangleq m = \text{count of tuples}$$

$$L[X_j] \triangleq \sum_{i=1}^{m} x_j^{(i)} = \text{sum of } X_j \text{ values}$$

$$Q[X_j, X_k] \triangleq \mathsf{Cofactor}[X_j, X_k] \triangleq \sum_{i=1}^{m} x_j^{(i)} x_k^{(i)}$$

L is thus a vector quantity, with an element corresponding to each model parameter, that is, one element per continuous-valued variable. Cofactor is a (square) matrix with one row and one column for each model parameter. A key insight of Schleich et al. [22] was that, with these aggregate definitions, the gradients for a single iteration of the batch gradient descent algorithm in equation (3.1) could be rewritten as

$$\frac{\partial}{\partial w_0} J(w) = w_0 + \frac{1}{C} \sum_{k=1}^{n} w_k L[X_k]$$

$$\frac{\partial}{\partial w_j} J(w) = \frac{w_0}{C} L[X_j] + \frac{1}{C} \sum_{k=1}^{n} w_k \mathsf{Cofactor}[X_j, X_k] \qquad j \in [n]$$

and that, with this rearrangement, the values of the query aggregates required at each convergence step of the batch gradient descent process would be independent of the values of the parameters at that step. This would enable computation of the aggregates (including the n-dimensional vector L and the $n \times n$ Cofactor matrix) to be performed only *once* over any given factorised query result; the batch gradient descent algorithm could then be performed directly on the precomputed aggregates, rather than repeatedly recomputing the query and the aggregates. Moreover, in the case of a dataset with n continuous-valued variables, the size of the cofactor matrix itself ($n \times n$) would be independent of the size m of the dataset. Two further insights of Schleich et al. [22] were, first, that computation of these aggregates would require only a single pass over any factorised query; in light of the size bounds from Proposition 2.19, for a specific query in the case of continuous-valued variables, this should enable the aggregates to be learned in time $\mathcal{O}(|\mathbf{D}|^{fhtw(Q)})$. And, second, that the Cofactor matrix is symmetric, meaning that only the upper half of the matrix need be computed.

The **F** algorithm is shown in Figure 3.1. Its structure follows closely the structure of the 'typical' algorithm over a factorisation shown in Figure 2.10 of Section 2.4.5. Using a similar structural analysis to that presented in Section 2.4.5, the overall computational complexity of the **F** algorithm depends on (i) the complexity of the computation 'f' executed once per union of the factorisation (that is, once per value symbol in the factorisation), in this case

```
foreach$B \in Schema[E_j]$ do {
    $L_E[B] = L_E[B] += L_{E_j}[B];$
    foreach $D \in Schema[E_j]$ do { $Q_E[B, D] += Q_{E_j}[B, D]$ }
}                                                                  (3.2)
```

and (ii) the computational complexity of the computation 'g' executed once per Cartesian product in the factorisation (that is, once per variable symbol in the factorisation *per* branch

```
F (Factorization E, varMap)
if (visited) return; switch E:

  ⟨A : a⟩
     |
     ×
    / \
 E_1 ··· E_k

    $C_E = 1$;
    foreach $j \in [k]$ do { F($E_j$); $C_E = C_E \cdot C_{E_j}$; }
    $L_E = a \cdot C_E$; $Q_E[A, A] = a \cdot L_E[A]$;
    foreach $j \in [k]$ do {
        foreach $B \in Schema[E_j]$ do {
            $L_E[B] = L_{E_j} \cdot C_E / C_{E_j}$
            $Q_E[A, B] = a \cdot L_{E_j}[B] \cdot C_E / C_{E_j}$
        }
        foreach $B, D \in Schema[E_j]$ s.t. $B < D$ do {
            $Q_E[B, D] = Q_{E_j}[B, D] \cdot C_E / C_{E_j}$
        }
        foreach $j < l \in [k], B \in Schema[E_l], B \in Schema[E_j]$ do {
            $Q_E[B, D] = L_{E_l}[B] \cdot L_{E_j}[D] \cdot C_E / (C_{E_l} \cdot C_{E_j})$;
        }
    }

     ∪
    / \
 E_1 ··· E_k

    $C_E = 0$; foreach $j \in [k], B, D \in Schema[E_j]$ do {
        $L_E[B] = 0, Q_E[B, D] = 0$;
    }
    foreach $j \in [k]$ do { F($E_j$); $C_E = C_E += C_{E_j}$; }
    foreach $j \in [k], B \in Schema[E_j]$ do {
        $L_E[B] = L_E[B] += L_{E_j}[B]$;
        foreach $D \in Schema[E_j]$ do { $Q_E[B, D] += Q_{E_j}[B, D]$ }
    }

visited = true;
```

Figure 3.1: **F** algorithm for computing the gradients for linear regression in one pass over a factorized join E. Algorithm taken from [22]

of the d-tree below that variable), in this case

```
foreach $B \in Schema[E_j]$ do {
    $L_E[B] = L_{E_j} \cdot C_E / C_{E_j}$
    $Q_E[A, B] = a \cdot L_{E_j}[B] \cdot C_E / C_{E_j}$
}
foreach $B, D \in Schema[E_j]$ s.t. $B < D$ do {
    $Q_E[B, D] = Q_{E_j}[B, D] \cdot C_E / C_{E_j}$
}
foreach $j < l \in [k], B \in Schema[E_l], B \in Schema[E_j]$ do {
    $Q_E[B, D] = L_{E_l}[B] \cdot L_{E_j}[D] \cdot C_E / (C_{E_l} \cdot C_{E_j})$;
}                                                                        (3.3)
```

Expression (3.2) is quadratic in the the size of $Schema[E_j]$. The first **foreach** term of (3.3) is linear in the size of $Schema[E_j]$, that is, in the size of the subtree of the d-tree rooted at variable E_j, and the second **foreach** term of (3.3) is quadratic in the size of $Schema[E_j]$. The final **foreach** term of (3.3) is quadratic in the size of $Schema[E_j]$, but *also* linear in $[k]$, that is, in the degree of branching of the d-tree at E_j. From equation (2.11) we thus have an intuitive computational complexity for **F** of

$$\begin{aligned}
&\mathcal{O}(\text{size of factorisation}(\mathcal{O}(f) + b \cdot \mathcal{O}(g)))\\
=&\mathcal{O}(\text{size of factorisation} \cdot (s^2 + b \cdot (b \cdot s^2))\\
=&\mathcal{O}(\text{size of factorisation} \cdot s^2 \cdot (1 + b^2))\\
=&\mathcal{O}(\text{size of factorisation} \cdot s^2 \cdot b^2) \qquad (3.4)
\end{aligned}$$

where b is again some measure of the 'average degree of branching' within the d-tree, and s is now some measure of the 'average *Schema* size' of the d-tree (that is, the average size of the traversal for a subtree of the d-tree). To note that b and s have an inverse relationship: the greater the degree of branching of a d-tree, the more 'shallow' the d-tree is, and the smaller the traversal of each of its subtrees. However, for any specific query (that is, for fixed b and s), the computational complexity of the algorithm is determined by the size of the factorisation, that is $\mathcal{O}(|\mathbf{D}|^{fhtw(Q)})$ from Proposition 2.19 (modulo log factors, if the time to construct the factorisation is included [22]).

Whilst b and s are not precisely defined for the general case, the fact that they can be determined for specific queries means that equation (3.4) nevertheless provides a useful baseline against which to compare improvements on the **F** algorithm.

Example 3.1. *Consider the (α-acyclic) query $Q_1 = R_1(X_1), R_2(X_2), \cdots, R_n(X_n)$ of Example 1.1. In the case of continuous-valued variables, the linear regression model has size n. For this query, the d-tree (shown in Figure 2.8) has maximal branching, with s = 1 and $b = n$. The computational complexity of **F** for this query is thus*

$$\begin{aligned}
&\mathcal{O}(|\mathbf{D}|^{fhtw(Q)} \cdot 1^2 \cdot n^2)\\
=&\mathcal{O}(|\mathbf{D}|^{fhtw(Q)} \cdot n^2)\\
=&\mathcal{O}(|\mathbf{D}|^1 \cdot n^2)\\
=&\mathcal{O}(nm \cdot n^2) = \mathcal{O}(n^3 m) \qquad (3.5)
\end{aligned}$$

that is, cubic overall in the size of the linear regression model n.

Example 3.2. *Consider the (also α-acyclic) single-relation query $Q_3 = R(X_1, X_2, \ldots, X_n)$, where $|R| = m$, in the case of continuous query variables, again with model size = number of query variables = n. Recall that dependent attributes (such as those within a single relation) lie on the same root to leaf path of the query's d-tree. The d-tree for query Q_3 therefore has no branching, thus $s = n/2$ (the average number of variables below each query variable in the single root to leaf path) and b = 1. In this case, the computational complexity of **F** is*

$$\begin{aligned}
&\mathcal{O}(|\mathbf{D}|^{fhtw(Q)} \cdot (n/2)^2 \cdot 1^2)\\
=&\mathcal{O}(|\mathbf{D}|^{fhtw(Q)} \cdot n^2)\\
=&\mathcal{O}(|\mathbf{D}|^1 \cdot n^2)\\
=&\mathcal{O}(nm \cdot n^2) = \mathcal{O}(n^3 m) \qquad (3.6)
\end{aligned}$$

again cubic overall in the size of the linear regression model n.

The original **F** algorithm handled only continuous-valued variables (it was extended for categorical variables in the **DC** algorithm of Khamis et al. [16]). However, in the case of queries Q_1 and Q_3 above, if each of the n relations had m distinct categorical values, the size of the model could be as large as $\mathcal{O}(m \cdot n) = \mathcal{O}(|\mathbf{D}|)$. In both cases, the resulting computational complexity of **F** would be

$$\mathcal{O}((nm) \cdot (nm)^2) = \mathcal{O}(n^3 \cdot m^3)$$

thus now cubic in the size of the database.

3.2 A New Improved Algorithm

3.2.1 Motivation

As seen in equations (3.5) and (3.6) above, using the **F** algorithm to compute the n^2 aggregates in the $n \times n$ Cofactor matrix for query Q with n continuous-valued variables over database **D** can take time

$$\mathcal{O}(|\mathbf{D}|^{fhtw(Q)} \cdot n^2) \tag{3.7}$$

However, the presence of one-hot encoded categorical variables introduces a distinct model parameter for each value taken by each categorical variable. In this case, even in the 'best' case of an acyclic query Q with $fhtw(Q) = 1$, the computational complexity of **F** can still be $\mathcal{O}(n^3)$. The ability to handle categorical variables more efficiently thus provides a key motivation for finding a better algorithm for computing the aggregates required to learn model parameters.

3.2.2 Rewriting the Gradients Using Ring Computation

The **F** algorithm uses only *semiring* computation (that is addition and multiplication) in its computation of the aggregates C, L and the $\mathsf{Cofactor}$ matrix. Returning to the gradients for the batch gradient descent method in (3.1), we now consider a slightly different set of aggregates, specifically:

$$\begin{aligned}
cnt &\triangleq m = \text{count of tuples} \\
\mathsf{S}_1[X_j] &\triangleq \sum_{i=1}^{m} x_j^{(i)} = \text{sum of } X_j \text{ values} \\
S_{1,w} &\triangleq \sum_{i=1}^{m} S_{1,w}^{(i)} \triangleq \sum_{i=1}^{m}\sum_{k=1}^{n} w_k x_k^{(i)} = \sum_{k=1}^{n} w_k \mathsf{S}_1[X_k] = \text{weighted sum of } X_j \text{ values} \\
\mathsf{Grad}[X_j] &\triangleq \sum_{i=1}^{m}\sum_{k=1}^{n} w_k x_j^{(i)} x_k^{(i)} = \text{weighted sum of paired products } X_j X_k
\end{aligned}$$

The aggregates cnt and S_1 correspond to the C and L aggregates used in **F**. $S_{1,w}$ is a new scalar quantity, and Grad is a vector, with an element corresponding to each model parameter, that is, one element per continuous-valued variable, and one element for each value taken by each categorical variable. With these new aggregate definitions, the gradients for a single iteration of the batch gradient descent algorithm from (3.1) become:

$$\begin{aligned}
\frac{\partial}{\partial w_0} J(w) &= w_0 + \frac{1}{cnt} S_{1,w} \\
\frac{\partial}{\partial w_j} J(w) &= \frac{w_0}{cnt}\mathsf{S}_1[X_j] + \frac{1}{cnt}\mathsf{Grad}[X_j] \qquad j \in [n]
\end{aligned}$$

The key insight of this project is that, using ring computation (specifically subtraction) it is possible to rewrite the formula for $\mathsf{Grad}[X_j]$ as follows:

$$
\begin{aligned}
\mathsf{Grad}[X_j] &= \sum_{i=1}^{m} \sum_{k=1}^{n} w_k x_j^{(i)} x_k^{(i)} \qquad j \in [n] \\
&= \sum_{i=1}^{m} \left(w_j x_j^{(i)} x_j^{(i)} + \left(\sum_{k=1}^{n} w_k x_j^{(i)} x_k^{(i)} \right) - w_j x_j^{(i)} x_j^{(i)} \right) \qquad j \in [n] \\
&= \sum_{i=1}^{m} \left(w_j x_j^{(i)} x_j^{(i)} + \left(\sum_{k=1}^{n} w_k x_j^{(i)} x_k^{(i)} - w_j x_j^{(i)} x_j^{(i)} \right) \right) \qquad j \in [n] \\
&= \sum_{i=1}^{m} \left(w_j x_j^{(i)} x_j^{(i)} + x_j^{(i)} \left(\sum_{k=1}^{n} w_k x_k^{(i)} - w_j x_j^{(i)} \right) \right) \qquad j \in [n] \\
&= \sum_{i=1}^{m} \Big(\underbrace{w_j x_j^{(i)} x_j^{(i)} + \underbrace{x_j^{(i)} \Big(\underbrace{S_{1,w}^{(i)} - w_j x_j^{(i)}}_{(1)} \Big)}_{(2)}}_{(3)} \Big) \qquad j \in [n] \qquad (3.8)
\end{aligned}
$$

For each value of i in (3.8), the quantity $\mathrm{S}_{1,w}^{(i)} = \sum_{k=1}^{n} w_k x_k^{(i)}$ is linear in n and need be computed only once; this can be done in time $\mathcal{O}(n)$. Moreover, once $\mathrm{S}_{1,w}^{(i)}$ has been computed, expression (1) can be computed in constant $\mathcal{O}(1)$ time with respect to n (a single multiplication and a single subtraction); expressions (2) and (3) can then each be computed in $\mathcal{O}(1)$ time (with a single multiplication, and two multiplications and an addition, respectively). Thus, once $\mathrm{S}_{1,w}^{(i)}$ has been calculated once in linear time, each of the n elements of Grad can be computed in constant time with respect to n.

The time to compute the full Grad vector is thus $\mathcal{O}(n) + n \cdot \mathcal{O}(1) = \mathcal{O}(n)$, compared with the $\mathcal{O}(n^2)$ time to compute the $\mathsf{Cofactor}$ matrix in the **F** algorithm. It is this ring-rewriting of Grad that forms the basis of the new algorithm, $\mathbf{F}^{\bigcirc}$.

3.2.3 The $\mathbf{F}^{\bigcirc}$ Algorithm

Like **F**, $\mathbf{F}^{\bigcirc}$ computes the aggregates over a materialised factorisation of the result of a join query; however, it now incorporates the rewritten ring calculation, and handles categorical values (the implementation in Chapter 4 also allows for *model selection* whereby query variables can be excluded completely as model features). Figure 3.2 on page 32 sets out the $\mathbf{F}^{\bigcirc}$ algorithm: Grad and S_1 are vectors with one entry per variable (with initial values 0) while *cnt* and $S_{1,w}$ are scalars. The model parameter corresponding to the value a_i taken by categorical variable A is denoted by $w_{A.a_i}$; the corresponding elements of S_1 and Grad are denoted by $S_1[A.a_i]$ and Grad$[A.a_i]$ respectively.

computational complexity of $\mathbf{F}^{\bigcirc}$ algorithm with continuous-valued variables Using the same structural analysis presented in Section 2.4.5 as used above for **F**, the overall computational complexity of this algorithm depends once again on the complexity of the the sections of computations 'f' and 'g'. In the case where all variables are continuous, the relevant computations for 'f' (executed once per union of the factorisation, that is, once per value symbol in the factorisation) are shown in (3.9) below. To note that the each $\sum_{j \in [k]}$ here is the equivalent of the **foreach** $j \in [k]$ of **typical-algo**; the complexity to consider is therefore that of each summand at the point where it is added to its respective sum.

$$
\begin{aligned}
&cnt = \sum_{j\in[k]} cnt^{(j)};\\
&\mathsf{S}_1 = \sum_{j\in[k]} \mathsf{S}_1^{(j)};\\
&\mathsf{S}_1[A] = \sum_{j\in[k]} a_j \cdot cnt^{(j)};\\
&S_{1,w} = \sum_{j\in[k]} S_{1,w}^{(j)} + w_A \cdot \mathsf{S}_1[A];\\
&\mathsf{Grad} = \sum_{j\in[k]} (w_A \cdot a_j \cdot \mathsf{S}_1^{(j)}[A] + \mathsf{Grad}^{(j)});\\
&\mathsf{Grad}[A] = \sum_{j\in[k]} a_j \cdot (w_A \cdot a_j \cdot cnt^{(j)} + S_{1,w}^{(j)}); \qquad (3.9)
\end{aligned}
$$

Now, each of the individual expressions being summed is of constant $\mathcal{O}(1)$ complexity at the point in the algorithm where it is added to the sum (for example, the summand $w_A \cdot a_j \cdot \mathsf{S}_1^{(j)}[A] + \mathsf{Grad}^{(j)}$ for Grad requires two multiplications and one addition of pre-computed quantities). Thus, the complexity of 'f' is $\mathcal{O}(1)$.

The relevant computations for 'g' (executed once per Cartesian product in the factorisation *per* branch of the d-tree below that variable) in this case are as shown in (3.14) below (where the $\sum_{j\in[k]}$ for $S_{1,w}$ is again the equivalent of the **foreach** $j \in [k]$ of **typical-algo**):

```
foreach $j \in [k]$ do {
    foreach $B \in Schema(\mathsf{Grad}^{(j)})$ do $\mathsf{S}_1[B] = \mathsf{S}_1^{(j)}[B] \cdot cnt/cnt^{(j)}$;        (3.10)
}
$S_{1,w} = \sum_{j\in[k]} S_{1,w}^{(j)} \cdot cnt/cnt^{(j)}$;        (3.11)
foreach $j \in [k]$ do {
    $S = S_{1,w} - S_{1,w}^{(j)} \cdot cnt/cnt^{(j)}$;        (3.12)
    foreach $B \in Schema(\mathsf{Grad}^{(j)})$ do {
        $\mathsf{Grad}[B] = (cnt \cdot \mathsf{Grad}^{(j)}[B] + S \cdot \mathsf{S}_1^{(j)}[B])/cnt^{(j)}$;        (3.13)
    }
}        (3.14)
```

Taking these in turn: expression(3.10) is linear in the size of the *Schema* of $\mathsf{Grad}^{(j)}$, that is, in the size of the subtree of the d-tree rooted at variable A. For each $j \in [k]$, expressions (3.11) and (3.12) are both $\mathcal{O}(1)$; finally,(3.13) is linear in the size of the *Schema* of $\mathsf{Grad}^{(j)}$. In the case of continuous variables, from (2.11), the intuitive complexity for $\mathbf{F}^{\bigcirc}$ is thus

$$
\begin{aligned}
&\mathcal{O}(\text{size of factorisation}(\mathcal{O}(f) + b \cdot \mathcal{O}(g)))\\
=&\mathcal{O}(\text{size of factorisation} \cdot (1 + s \cdot b))\\
=&\mathcal{O}(\text{size of factorisation} \cdot s \cdot b) \qquad (3.15)
\end{aligned}
$$

where b is again some measure of the 'average degree of branching' within the d-tree, and s is again some measure of the 'average *Schema* size' of the d-tree (that is, the average size of the traversal for a subtree of the d-tree). Revisiting the two specific examples considered above for $\mathbf{F}$, but now for $\mathbf{F}^{\bigcirc}$:

Example 3.3. *Consider the (α-acyclic) query $Q_1 = R_1(X_1), R_2(X_2), \cdots, R_n(X_n)$. In the case of continuous-valued variables, the linear regression model has size n. For this query, the* *d-tree (shown in Figure 2.8) has maximal branching, with $s = 1$ and $b = n$. The complexity of $\mathbf{F}^{\bigcirc}$ for this query is thus*

$$\begin{aligned} &\mathcal{O}(|\mathbf{D}|^{fhtw(Q)} \cdot n \cdot 1) \\ =&\mathcal{O}(|\mathbf{D}|^{1} \cdot n) = \mathcal{O}(|\mathbf{D}| \cdot n) \end{aligned} \tag{3.16}$$

Example 3.4. *Consider the (also α-acyclic) single-relation query $Q_3 = R(X_1, X_2, \dots, X_n)$, where $|R| = m$, again with model size n. Recall that dependent attributes (such as those within a single relation) lie on the same root to leaf path of the query's d-tree. The d-tree for query Q_3 therefore has no branching, thus $s = n/2$ and $b = 1$. In this case, the complexity of $\mathbf{F}^{\bigcirc}$ is*

$$\begin{aligned} &\mathcal{O}(|\mathbf{D}|^{fhtw(Q)} \cdot 1 \cdot n/2) \\ =&\mathcal{O}(|\mathbf{D}|^{1} \cdot n) = \mathcal{O}(|\mathbf{D}| \cdot n) \end{aligned} \tag{3.17}$$

Comparing with equations (3.5) and (3.6), $\mathbf{F}^{\bigcirc}$ can thus be seen to offer an asymptotic complexity improvement on $\mathbf{F}$ in the case of continuous variables.

Time-complexity of $\mathbf{F}^{\bigcirc}$ with categorical variables In the case of categorical variables, the relevant computations for the 'f' section of the $\mathbf{F}^{\bigcirc}$ algorithm are now:

```
$cnt = \sum_{j \in [k]} cnt^{(j)};$
$\mathsf{S}_1 = \sum_{j \in [k]} \mathsf{S}_1^{(j)};$
**foreach** $j \in [k]$ **do** $\mathsf{S}_1[A.a_j] = cnt^{(j)};$
$S_{1,w} = \sum_{j \in [k]} S_{1,w}^{(j)} + \sum_{j \in [k]} w_{A.a_j} \cdot \mathsf{S}_1[A.a_j];$
$\mathsf{Grad} = \sum_{j \in [k]} (w_{A.a_j} \cdot 1 \cdot \mathsf{S}_1^{(j)}[A.a_j] + \mathsf{Grad}^{(j)});$
**foreach** $j \in [k]$ **do** $\mathsf{Grad}[A.a_j] = w_{A.a_j} \cdot cnt^{(j)} + S_{1,w}^{(j)};$
```

Again each of the expressions being summed is $\mathcal{O}(1)$ with respect to the size of the database and the size of *Schema*$[A]$; thus 'f' again has complexity $\mathcal{O}(1)$. The relevant computations for 'g' are as before:

```
**foreach** $j \in [k]$ **do** {
    **foreach** $B \in Schema(\mathsf{Grad}^{(j)})$ **do** $\mathsf{S}_1[B] = \mathsf{S}_1^{(j)}[B] \cdot cnt/cnt^{(j)};$        (3.18)
}
$S_{1,w} = \sum_{j \in [k]} S_{1,w}^{(j)} \cdot cnt/cnt^{(j)};$        (3.19)
**foreach** $j \in [k]$ **do** {
    $S = S_{1,w} - S_{1,w}^{(j)} \cdot cnt/cnt^{(j)};$        (3.20)
    **foreach** $B \in Schema(\mathsf{Grad}^{(j)})$ **do** {
        $\mathsf{Grad}[B] = (cnt \cdot \mathsf{Grad}^{(j)}[B] + S \cdot \mathsf{S}_1^{(j)}[B])/cnt^{(j)};$        (3.21)
    }
}
```

Expressions (3.19) and (3.20) are both $\mathcal{O}(1)$, and expressions (3.18) and (3.21) are again both linear in the size of the *Schema* of $\mathsf{Grad}^{(j)}$. However, rather than its *Schema* size corresponding to the number of query variables, $\mathsf{Grad}^{(j)}$ now contains an element for every variable in the subtree of the d-tree rooted at variable A, *and* for every value taken by any of those descendent variables that are categorical. An intuitive overall complexity for $\mathbf{F}^{\bigcirc}$ is now therefore

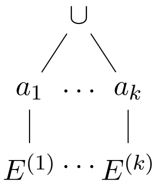
$$\mathcal{O}(\text{size of factorisation} \cdot (1 + s_c \cdot b)) \tag{3.22}$$

where b is again some measure of the 'average degree of branching' within the d-tree, and s_c is now the average size of a subtree within the d-tree, weighted according to the number of values taken by any categorical variables within that subtree.

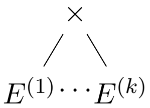
Two points to note: first, a given categorical variable has greater (negative) impact on the complexity of $\mathbf{F}^{\bigcirc}$ the more times it appears in a distinct subtree of the d-tree. In general, therefore, the further a categorical variable is from the root of the d-tree, the greater its potential impact on the complexity of $\mathbf{F}^{\bigcirc}$. Second, we know that for a given query (and d-tree), the complexity of $\mathbf{F}^{\bigcirc}$ is linear in s_c which, in turn, is no greater than the size of the linear regression model. The $\mathbf{F}^{\bigcirc}$ algorithm thus offers again an asymptotic complexity improvement over $\mathbf{F}$.

There is a significant caveat however! The improved complexity of $\mathbf{F}^{\bigcirc}$ for calculating the aggregates for a *single* iteration of the batch gradient descent algorithm comes at a cost. Specifically, the computation of the aggregates is no longer decoupled from the iterative steps of the batch gradient descent algorithm, each of which requires a separate computational pass over the factorised query to compute a fresh set of aggregates ($S_{1,w}$ and Grad) based on the latest values of the parameters. On the assumption, however, that the number of iterative steps $\ll$ the size of the model in the case of categorical variables, $\mathbf{F}^{\bigcirc}$ may still deliver improved performance overall.

```
F^○ (Factorization E, varMap)
switch $E$:
```



```
$A = var(E)$;    keyMap $= \pi_{key(A)}$(varMap);
if $(key(A) \neq anc(A))$ {
    $(cnt, S_{1,w}, \mathsf{S}_1, \mathsf{Grad}) = cache$[keyMap];  if $(cnt \neq 0)$ break;
}
if $(\forall j \in [k] : E^{(j)} = \emptyset)$ then foreach $j \in [k]$ do $cnt^{(j)} = 1$;
else foreach $j \in [k]$ do {
    $(cnt^{(j)}, S^{(j)}_{1,w}, \mathsf{S}^{(j)}_1, \mathsf{Grad}^{(j)}) = \mathbf{F}^{\bigcirc}(E^{(j)}, \text{varMap} \times \{(A : a_j)\})$;
}
$cnt = \sum_{j\in[k]} cnt^{(j)}$;
$\mathsf{S}_1 = \sum_{j\in[k]} \mathsf{S}^{(j)}_1$;
if (A is a categorical variable) then {
    foreach $j \in [k]$ do $\mathsf{S}_1[A.a_j] = cnt^{(j)}$;
}
else $\mathsf{S}_1[A] = \sum_{j\in[k]} a_j \cdot cnt^{(j)}$;
if (A is a categorical variable) then {
    $S_{1,w} = \sum_{j\in[k]} S^{(j)}_{1,w} + \sum_{j\in[k]} w_{A.a_j} \cdot \mathsf{S}_1[A.a_j]$;
    $\mathsf{Grad} = \sum_{j\in[k]}(w_{A.a_j} \cdot 1 \cdot \mathsf{S}^{(j)}_1[A.a_j] + \mathsf{Grad}^{(j)})$;
    foreach $j \in [k]$ do $\mathsf{Grad}[A.a_j] = w_{A.a_j} \cdot cnt^{(j)} + S^{(j)}_{1,w}$;
}
else {
    $S_{1,w} = \sum_{j\in[k]} S^{(j)}_{1,w} + w_A \cdot \mathsf{S}_1[A]$;
    $\mathsf{Grad} = \sum_{j\in[k]}(w_A \cdot a_j \cdot \mathsf{S}^{(j)}_1[A] + \mathsf{Grad}^{(j)})$;
    $\mathsf{Grad}[A] = \sum_{j\in[k]} a_j \cdot (w_A \cdot a_j \cdot cnt^{(j)} + S^{(j)}_{1,w})$;
}
if $(key(A) \neq anc(A))$   cache[keyMap] $= (cnt, S_{1,w}, \mathsf{S}_1, \mathsf{Grad})$;
```



```
foreach $j \in [k]$ do {
    $(cnt^{(j)}, S^{(j)}_{1,w}, \mathsf{S}^{(j)}_1, \mathsf{Grad}^{(j)}) = \mathbf{F}^{\bigcirc}(E^{(j)}, \text{varMap})$;
}
$cnt = \prod_{j\in[k]} cnt^{(j)}$;
foreach $j \in [k]$ do {
    foreach $B \in Schema(\mathsf{Grad}^{(j)})$ do $\mathsf{S}_1[B] = \mathsf{S}^{(j)}_1[B] \cdot cnt/cnt^{(j)}$;
}
$S_{1,w} = \sum_{j\in[k]} S^{(j)}_{1,w} \cdot cnt/cnt^{(j)}$;
foreach $j \in [k]$ do {
    $S = S_{1,w} - S^{(j)}_{1,w} \cdot cnt/cnt^{(j)}$;
    foreach $B \in Schema(\mathsf{Grad}^{(j)})$ do {
        $\mathsf{Grad}[B] = (cnt \cdot \mathsf{Grad}^{(j)}[B] + S \cdot \mathsf{S}^{(j)}_1[B])/cnt^{(j)}$;
    }
}
return $(cnt, S_{1,w}, \mathsf{S}_1, \mathsf{Grad})$;
```

Figure 3.2: $\mathbf{F}^{\bigcirc}$ algorithm for computing the gradients for linear regression in one pass over a factorized join E.

Chapter 4

Implementation

Chapter 3 presented a a new algorithm, $\mathbf{F}^{\bigcirc}$, for computing query aggregates over the training dataset for a linear regression model. The $\mathbf{F}^{\bigcirc}$ algorithm takes as its input a materialised factorisation of the query join; the output of each application of the algorithm is the set of query aggregates required for a single set of gradients of the least-squares objective function, that is, for a single iteration (*convergence step*) of the batch gradient descent process.

Implementation of the end-to-end batch gradient descent process therefore involves three stages: (i) materialisation of the factorised join; (ii) implementation of the $\mathbf{F}^{\bigcirc}$ algorithm to compute the required query aggregates; and (iii) embedding the $\mathbf{F}^{\bigcirc}$ computations within the batch gradient descent iterative process. This chapter describes the practical implementation in `C++` of each these stages, including some of the classes, methods, and individual code fragments used. Chapter 5 then describes the experiments carried out to compare the practical performance of this implementation against its state of the art competitors.

4.1 Deliverables and the C++ Codebase Used

The implementation of $\mathbf{F}^{\bigcirc}$ within the batch gradient descent process modifies and extends an existing `C++` codebase for the construction of factorised database representations and linear regression models. The resulting modified structure of the various classes and data sources is shown in Figure 4.1.

The central deliverable of the implementation is the class `CategoricalRegressionOverJoin`, the methods of which implement the $\mathbf{F}^{\bigcirc}$ algorithm for training datasets involving arbitrary combinations of continuous and categorical model features (and also allow for individual query variables to be excluded from the features of the model). The full code listing for the new class `CategoricalRegressionOverJoin.cpp` is included as Appendix A and in electronic format with this dissertation.

Other classes (`DTreeNode`, `CategoricalFactorisedJoin`, and the `Union` struct) are modifications or extensions of existing classes, for example, to support the introduction of categorical (and excluded) features into the implementation, whilst the `DTree` and `Launcher` classes have been used with only very minimal changes. The code of selected classes that have been added to the `CategoricalFactorisedJoin` class, and which are discussed below, is included as Appendix B and in electronic format with this dissertation. The full `C++` codebase is held in a private GitHub repository; if you would like access to this, please contact Professor Dan Olteanu or one of his team.

4.2 Materialised Factorisation of the Join Query

4.2.1 Data Structures Used for D-Tree and Factorisation

Recall that the branching structure of the factorised representation of a query is determined by the branching structure of the underlying d-tree for the query. Algorithms that iterate over a

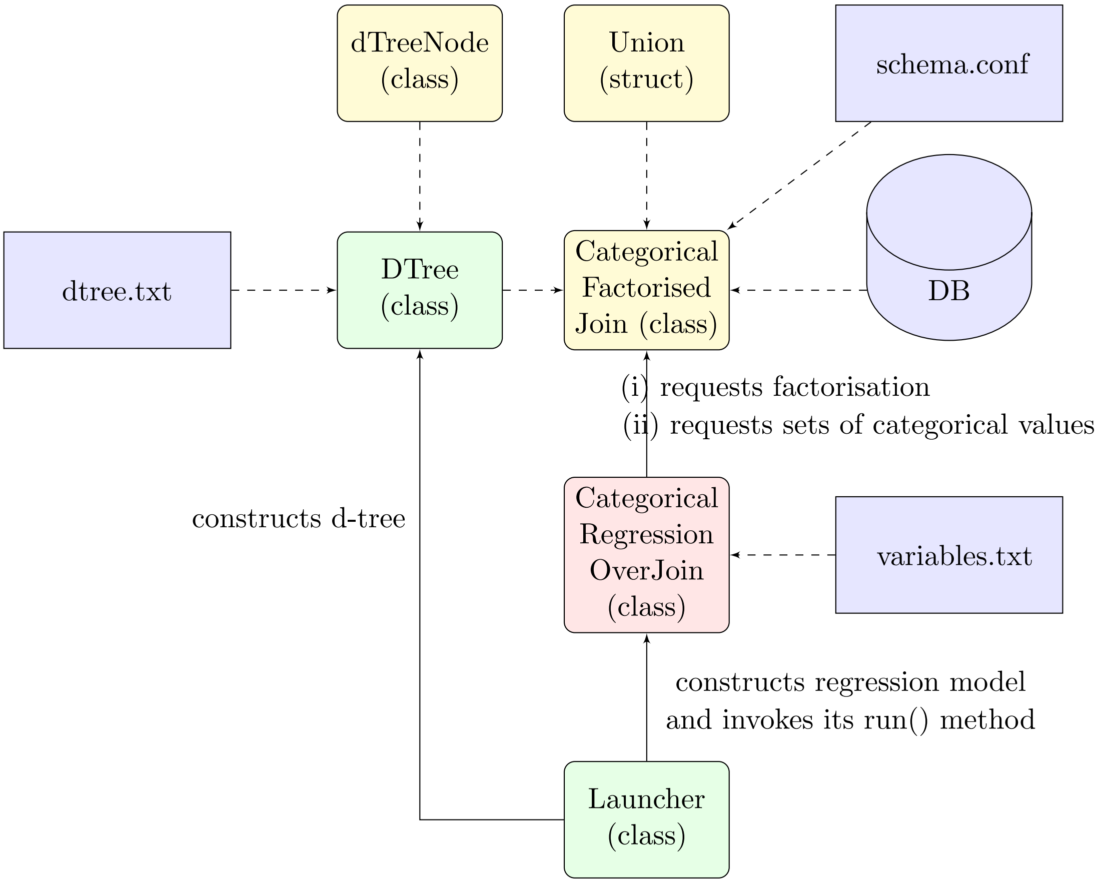

dTreeNode (class)
Union (struct)
schema.conf
dtree.txt
DTree (class)
Categorical Factorised Join (class)
DB
(i) requests factorisation
(ii) requests sets of categorical values
constructs d-tree
Categorical Regression OverJoin (class)
variables.txt
constructs regression model and invokes its run() method
Launcher (class)

Figure 4.1: Class structure and data inputs to the $\mathbf{F}^{\bigcirc}$ implementation. Classes in green are used with minimal changes, classes in yellow with minor modifications (e.g. to accommodate catetorical features), and the `CategoricalRegressionOverJoin` class in red is new for the implementation.

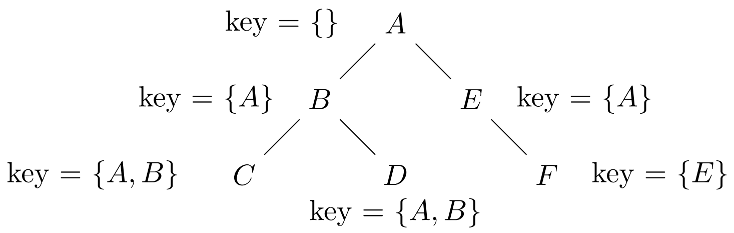


Figure 4.2: d-tree for natural join $Q_2 = R_1(A, B, C), R_2(A, B, D), R_3(A, E), R_4(E, F)$

materialised factorisation, such as $\mathbf{F}$ and $\mathbf{F}^{\bigcirc}$, therefore require both the factorisation and the d-tree to be materialised in memory. Methods for computing aggregates over a factorisation take as their arguments both a factorisation (or a subtree of a factorisation) and the corresponding d-tree (or subtree of the d-tree).

The d-tree is materialised as a tree of doubly linked nodes, each of which is an instance of the `DTreeNode` class, corresponding to a single node (that is, query variable) of the d-tree. The information stored in each `DTreeNode` object includes the name of the query variable together with parent, child and sibling information to aid navigation through the tree.

The materialised factorisation of the query dataset also uses nodes, this time instances of the `Union` struct. Each `Union` instance corresponds to a single union in the factorisation (that is, a union of singletons $\langle A : a \rangle$ for some query variable A). Figure 4.3 shows the correspondence between the factorisation and the `Union` instances in its materialisation for the query $Q_2 = R_1(A, B, C), R_2(A, B, D), R_3(A, E), R_4(E, F)$ where the tuples in each relation are as shown in Figure 2.2(a). Notice that each `Union` instance in the materialised factorisation has one child `Union` instance per value $\langle A : a \rangle$ and per branch at the query variable A in the d-Tree (shown in Figure 4.2). This corresponds precisely to the branching structure of the $\mathbf{F}^{\bigcirc}$ algorithm, where computations are performed once per value, per branch of the d-tree at that value's variable.

The leapfrog join (described in Section 2.5) is used for two purposes in the implementation of $\mathbf{F}^{\bigcirc}$. First, it forms the basis of an recursive method to create a factorised representation of the query join, using the structure of the query's d-tree to define the *variable order* for the join (and thus the branching structure of the factorisation itself). The `run()` method from an existing `C++` class in the codebase (`FactorizedJoin`) was reused as the `run()` method of a new class `CategoricalFactorizedJoin`. This is invoked by the `CategoricalRegressionOverJoin` class, returning the root node of the materialised factorisation, over which the recursive $\mathbf{F}^{\bigcirc}$ computations are performed.

4.2.2 Construction of the Materialised Factorisation

The factorised representation returned by the `run()` method is similar to that of Figure 4.3 (for example, it employs caching) but with two key additions. Standard relational algebra operates under *set semantics*, and a relation may not contain duplicate tuples. In practice, however, relational database systems (and SQL) operate under *bag semantics*, permitting duplicate values. Moreover, supervised learning tasks such as linear regression are sensitive to these: the values of the learned model parameters depend on the frequencies of training dataset items as well as their values. Thus the first addition to the materialised factorisation is that each `Union` instance records not only a set of *values* $\langle A : a \rangle$ but also the *frequency* with which each value occurs in the underlying query.

The second addition is that, given the recursive nature of the leapfrog join implementation, it is straightforward to incorporate the computation of the count values (*cnt*) from the $\mathbf{F}^{\bigcirc}$ algorithm into the construction of the factorisation itself, and this has been done. This does not affect the computational complexity of either algorithm: it simply transfers a computational step from the $\mathbf{F}^{\bigcirc}$ algorithm to the join algorithm. In the case of query $Q_2 = R_1(A, B, C), R_2(A, B, D), R_3(A, E), R_4(E, F)$, and with these two additions, the factori-

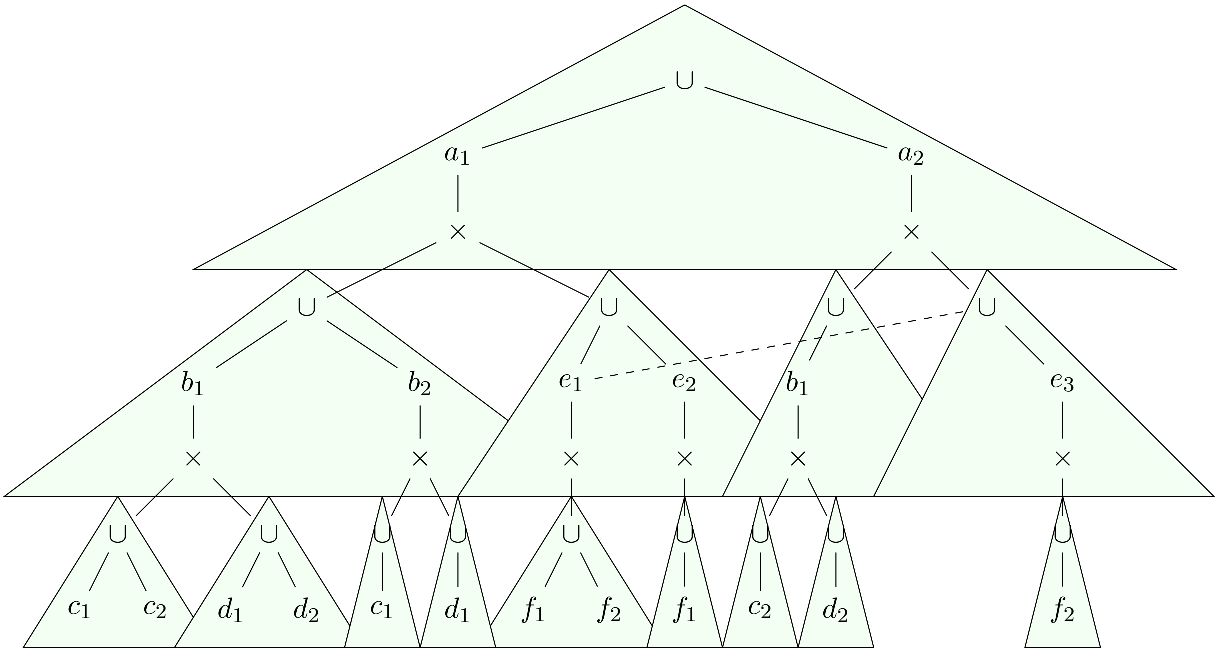


Figure 4.3: Factorisation of $Q_2 = R_1(A,B,C), R_2(A,B,D), R_3(A,E), R_4(E,F)$, where the tuples in each relation are as shown in Figure 2.2(a), showing the correspondence with `Union` instances in the materialised factorisation.

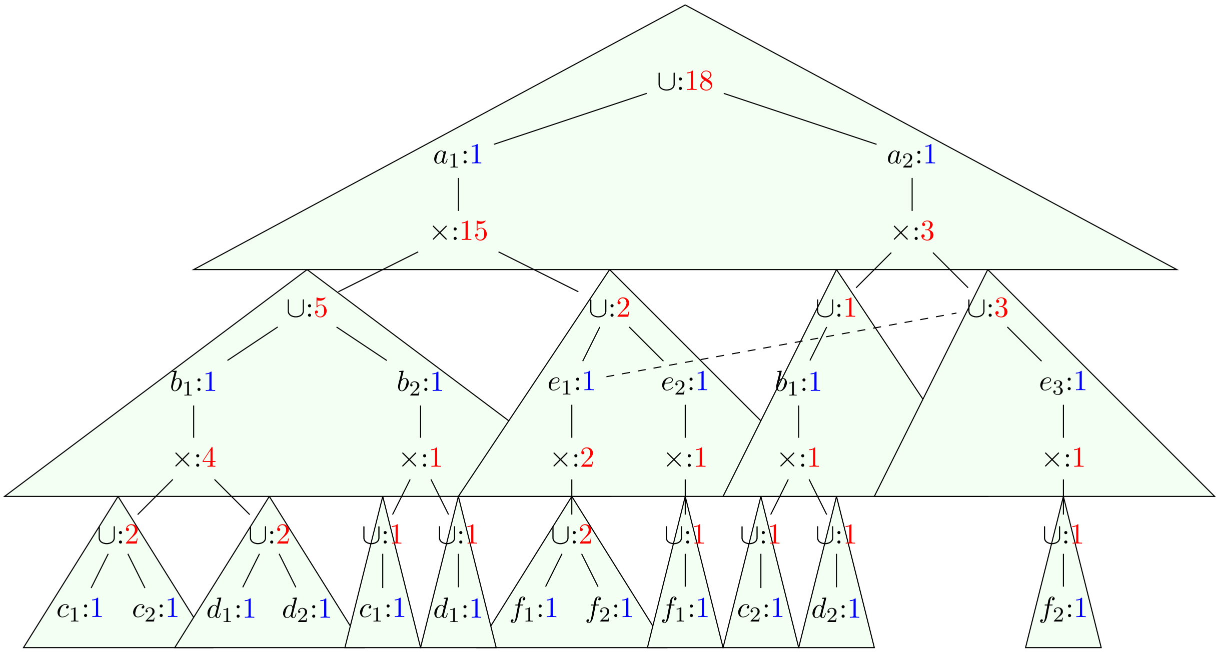


Figure 4.4: Factorisation of $Q_2 = R_1(A,B,C), R_2(A,B,D), R_3(A,E), R_4(E,F)$, where the tuples in each relation are as shown in Figure 2.2(a), using caching, including frequencies, together with *cnt* values for each union and Cartesian product.

```
CategoricalRegressionOverJoin::getCategoricalValues() {
        CategoricalFactorisedJoin::sortDataForCategoricalValues();
        CategoricalFactorisedJoin::getCategoricalValues() {
                CategoricalFactorisedJoin::runCategoricalValuesFinder() {
                        // implementation of leapfrog join on the categorical variable
                        CategoricalFactorisedJoin::leapfroggingJoinSingleAttribute() {
                }
        }
}
```

Figure 4.5: High-level structure of the methods of the CategoricalFactorisedJoin class invoked by the getCategoricalValues() method of the CategoricalRegressionOverJoin class to determine the values taken by each categorical query variable (method detail omitted). The code for the methods added to the CategoricalFactorisedJoin class is included in Appendix B.

sation over which the implementation of $\mathbf{F}^{\bigcirc}$ operates is now as shown in Figure 4.4. Notice that, as per the $\mathbf{F}^{\bigcirc}$ algorithm, the *cnt* value at a union is the semiring sum of the *cnt* values of the child Cartesian products below that union (or, where there is no Cartesian product, the count of any leaf values below that union). Conversely, the *cnt* value at a Cartesian product is the semiring product of the *cnt* values of the child unions below that Cartesian product.

The second use of leapfrog join in the implementation is to compute the set of values that are taken by each categorical query variable (recall that each value taken by a categorical query variable maps to a distinct parameter in the linear regression model). This is achieved by sorting and then joining the query's relations only on that categorical variable. Four new methods of the `CategoricalFactorizedJoin` class are invoked by the `getCategoricalValues()` method of the `CategoricalRegressionOverJoin` class, the high-level structure of which is shown in Figure 4.5 (the code for these methods is listed in Appendix B).

Together these four methods return the full set of values taken by each categorical variable in the query. This information across all categorical variables, in turn, determines the number of model parameters required, and thus the size of the the associated vectors of aggregates required for each batch gradient descent iteration. It also allows a mapping to be constructed from each categorical value to an index position within those vectors (described in more detail in section 4.3.1 below).

4.3 Implementation of $\mathbf{F}^{\bigcirc}$ Aggregate Computations

4.3.1 Structure of Aggregate Vectors

Each application of the $\mathbf{F}^{\bigcirc}$ algorithm computes the set of aggregates *cnt*, S_1, $S_{1,w}$ and Grad over the whole materialised factorisation of the query dataset. Considering first the case where all features are continuous, *cnt* and $S_{1,w}$ are scalars, and S_1 and Grad are vectors, each with one element corresponding to each model parameter, that is per query variable. Each of these aggregates is computed recursively over the factorisation; the implementation of $\mathbf{F}^{\bigcirc}$ does this by storing the aggregate values computed at each union of the factorisation in the corresponding `Union` instance. Thus four instance variables are added to the `Union` struct: *cnt* and $S_{1,w}$ are stored as `doubles`, and the vector quantities S_1 and Grad are stored as simple `C++ array` structures.

The ordering of the elements within the S_1 and Grad vectors is, however, crucial to the implementation of $\mathbf{F}^{\bigcirc}$. At the point where S_1 and Grad are being computed for a given `Union` instance corresponding to a union of $\langle A : a \rangle$ values, the only *non-zero* elements of S_1 and Grad are those corresponding to the query variables in the subset of the d-tree rooted at query variable A (in the listing of the $\mathbf{F}^{\bigcirc}$ algorithm in Figure 3.2, this is termed the *Schema* of the variable). At a `Union` instance in the factorisation corresponding to a leaf variable of the d-tree, the S_1 and Grad vectors both contain only a single non-zero element (corresponding to that leaf query variable). As the computation proceeds up the factorisation to its root (corresponding to the root of the d-tree), the size of S_1 and Grad increases from 1 to the number of variables in the query. A 'good' choice of ordering for the elements of S_1 and Grad can therefore make for simpler and more elegant computations within the algorithm, and avoid the need for repeated computations on zero vector elements (in a worst case, potentially reducing the complexity gains promised by the algorithm).

The ordering identified as 'best' for this implementation corresponds to the post-order (left-right-root) traversal of the d-tree; for $Q_2 = R_1(A, B, C), R_2(A, B, D), R_3(A, E), R_4(E, F)$ with d-tree shown in Figure 4.2, the ordering is thus $[C, D, B, F, E, A]$. The specific advantages of this ordering are as follows: first, the *Schema* of any query variable in the d-tree forms a contiguous subset of this traversal (for example, $[C, D, B]$ for query variable B, or $[F, E]$ for query variable E). Second, the *Schemas* of the children of a query variable in the d-tree form a contiguous set of disjoint subsets of the traversal (for example, the children of A are B and E with *Schemas* $[C, D, B], [F, E]$ respectively). Finally, just as the structure of a d-tree can be determined by static analysis of the query (irrespective of the specific content of the database), the size and individual element positions of the S_1 and Grad vectors for any given `Union` instance can be obtained from a simple traversal of the relevant subtree of the d-tree.

With this choice of ordering, the S_1 and Grad vectors for any given `Union` instance can be constructed to hold *only* elements corresponding to the schema of that `Union` instance's query variable. Intuitively, as the S_1 and Grad aggregates are processed at a Cartesian product in the factorisation, the variables in the *Schema* of each child's vector of aggregates simply take the next available index positions in the parent's vector of aggregates; the parent's own query variable then takes the final position in its own vector of aggregates. This also ensures that, at each union of the factorisation, any aggregates being combined always have their elements in consistent positions.

The computations performed by $\mathbf{F}^{\bigcirc}$ on the S_1 and Grad vectors also involve the latest values of the model parameters $\mathsf{w} = (w_1, w_2, \ldots, w_n)$, a vector of constant size stored as a `C++ array` of type `double`. Consistent with the indexing chosen for the vectors of aggregates, in the case of continuous features, the index position of each model parameter within that vector is set as the position of its associated query variable in the traversal of the full d-tree. When carrying out computations involving the aggregate vectors for any given query variable A, to ensure that the elements of those vectors are always combined with the correct model parameters and the correct elements of the aggregate vectors for other query variables, three pieces of information are therefore required for the variable A:

(i) the number of elements in the aggregate vectors corresponding to variable A, that is, the size of the traversal of the subtree of the d-tree rooted at A.

(ii) the index position in the post-order traversal of the *whole* d-tree of the first element in the post-order traversal of the subtree of the d-tree rooted at A, that is, the parameter corresponding to the 'first' element of A's own aggregate vectors.

(iii) a mapping from any given query variable in the subtree of the d-tree rooted at A to that variable's position in the post-order traversal of that subtree, including the index for the variable A itself.

Each of these items of information is a property purely of the d-tree, irrespective of the data in the query. For implementation purposes, additional instance variables have therefore been added to the DTreeNode class:

- for (i) `unsigned int _numOfDescendentAggr`;
- for (ii) `unsigned int _indexOfNodeInRootAggregates`; and
- for (iii) the traversal itself `vector<unsigned int> _traversal` and the required mapping of variables to indices `map<unsigned int, unsigned int> _indexOfAttrInNodeAggregates`.

In the case of categorical features, exactly the same variable ordering can be used within the S_1 and Grad vectors; however, each value taken by a categorical query variable now results in a separate model parameter. The vector of model parameters and final aggregate vectors S_1 and Grad returned by the $\mathbf{F}^{\bigcirc}$ algorithm now include one index position for each value taken by each categorical feature. As the $\mathbf{F}^{\bigcirc}$ computations proceed up the tree, however, there is no guarantee that computations for different `Union` instances will see the same categorical values in the same order! Two approaches are possible here: one more 'dynamic', one more 'static'. In the more dynamic approach, the S_1 and Grad vectors for each `Union` instance start with a single element for each categorical feature; as more values for that categorical variable are seen by the computations over that object, S_1 and Grad expand accordingly. An advantage of this approach is that S_1 and Grad never contain redundant zeros. However, as S_1 and Grad are processed by the computations at each Cartesian products and union of the factorisation, a complex process of repeated resizing and re-indexing is required to maintain consistent element positions within each of the aggregate vectors (and within the vector of model parameters itself) when combining aggregate vectors higher up the factorisation. The more 'static' approach is to determine in advance the values taken by each categorical variable (this can be done in a similar way to constructing the full factorisation, recording only the values of the categorical variable). Knowing the possible categorical values determines the number of elements that will be required in the S_1 and Grad vectors; it also enables a fixed mapping from each categorical value to an index position within each vector, resulting in more computations on zero vector elements, but avoiding the computational effort of repeated re-sizing and re-indexing.

The 'static' approach has been taken in the implementation, with a further instance variable added to the DTreeNode class to hold the number of categorical variables for that query variable (now determined, as noted above, prior to the execution of the $\mathbf{F}^{\bigcirc}$ computations). The mapping between the value of a categorical feature and its index positions within a sub-vector of aggregates for that particular feature is the same for *all* vectors of aggregates where that query variable has a corresponding index position (for example, in the case of query Q_2 the variable C has an index position ($= 0$) in the aggregate vectors for query variables A, B, and C. Rather than being stored in a specific DTreeNode, this information is therefore stored as an instance variable of the `CategoricalRegressionOverJoin` class, from where it can be made available to all relevant computations (as a map `_catAggrToOverallAggrMapping`).

4.3.2 Methods Performing the Core $\mathbf{F}^{\bigcirc}$ Computations

The key methods to perform the end-to-end batch gradient descent process including the $\mathbf{F}^{\bigcirc}$ computations are all invoked from the `run()` method of the `CategoricalRegressionOverJoin` class, the high-level structure of which is shown in Figure 4.6. **The four new methods which actually implement the $\mathbf{F}^{\bigcirc}$ computations are `computeMultiplicity`, `computeS1`, `computeS1w`, and `computeGrad`, the full code for which is included in Appendix A.**

Recall the aggregates that are being computed (including the ring rewriting of Grad) are as follows:

$$cnt \triangleq m, \qquad \mathsf{S}_1[X_j] \triangleq \sum_{i=1}^{m} x_j^{(i)}$$

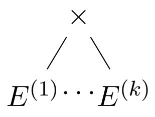
$$S_{1,w} \triangleq \sum_{i=1}^{m} S_{1,w}^{(i)} \triangleq \sum_{i=1}^{m}\sum_{k=1}^{n} w_k x_k^{(i)}$$

$$\mathsf{Grad}[X_j] \triangleq \sum_{i=1}^{m}\left(w_j x_j^{(i)} x_j^{(i)} + x_j^{(i)}\left(S_{1,w}^{(i)} - w_j x_j^{(i)}\right)\right)$$

and that the *cnt* quantities have been computed for each `Union` instance in the factorisation during its construction. Notice also that the values of S_1 are independent of the values of the model parameters at any point in the batch gradient descent process. The computation of S_1 can thus be decoupled from the batch gradient descent iterations and can be performed

```
CategoricalRegressionOverJoin::run() {
        loadFeatures();
        getCategoricalValues();
        computeAggregateVectorInfo(dTreeRoot);
        _joinEngine->run();
        computeParameters() {
                computeMultiplicity(rootNode, _dTree->_root, 1);
                computeS1(rootNode, _dTree->_root);
                // choose initial (random) model parameters
                optimize() {
                        // first convergence step of batch gradient descent
                        computeGradient() {
                                computeS1w(rootNode, _dTree->_root)
                                computeGrad(rootNode, _dTree->_root)
                        }
                        // subsequent convergence steps of batch gradient descent
                        while(!converged){
                                //update model parameters in direction of gradient
                                computeGradient() {
                                        computeS1w(rootNode, _dTree->_root)
                                        computeGrad(rootNode, _dTree->_root)
                                }
                        }
                }
        }
}
```

Figure 4.6: High level structure of the methods invoked from the run() method of the CategoricalRegressionOverJoin class to compute a complete set of linear regression model parameters (method detail omitted). Full code for this class is given in Appendix A.



```
$S_{1,w} = \sum_{j \in [k]} S_{1,w}^{(j)} \cdot cnt/cnt^{(j)}$;
foreach $j \in [k]$ do {
    $S = S_{1,w} - S_{1,w}^{(j)} \cdot cnt/cnt^{(j)}$;
    foreach $B \in Schema(\mathsf{Grad}^{(j)})$ do {
        $\mathsf{Grad}[B] = (cnt \cdot \mathsf{Grad}^{(j)}[B] + S \cdot \mathsf{S}_1^{(j)}[B])/cnt^{(j)}$;
    }
}
```

Figure 4.7: Section of $\mathbf{F}^{\bigcirc}$ algorithm with ring computation; code for the higlighted computational step is shown in Figure 4.9

over the factorisation only once, using the method `computeS1`. There is another component of the $\mathbf{F}^{\bigcirc}$ computations that can be decoupled in this way: from the algorithm listing in Figure 3.2, it can be seen that the computations for S_1, $S_{1,w}$ and Grad each involve a 'multiplier' factor $cnt/cnt^{(j)}$, which is the same for each `Union` instance in the factorisation, and is also independent of the particular values of the model parameters at a given convergence step of the batch gradient descent process. Its computation is also performed once over the whole factorisation by the `computeMultiplicity` method. This leaves the $S_{1,w}$ and Grad quantities: these must be computed afresh each time the model parameters are updated, hence their inclusion within the loop structure of the `optimize` method.

A consequence of this decoupling is that computation of a full set of aggregates using the $\mathbf{F}^{\bigcirc}$ algorithm now involves more than one pass over the factorisation, specifically one pass for the computation of each aggregate S_1, $S_{1,w}$ and Grad. This does not affect the complexity of the algorithm (there are now fewer computational steps within each pass); aside from removing redundant identical computations of S_1 and $cnt/cnt^{(j)}$, it also makes the code for each of the individual S_1, $S_{1,w}$ and Grad methods simpler and more maintainable. As will be seen in Chapter 5 however, experimental results on large-scale datasets in fact revealed that this design choice had a significant negative impact on practical performance.

Ring computation The key contribution of the $\mathbf{F}^{\bigcirc}$ algorithm of Figure 3.2 is the use of ring computation, specifically in the computational steps for Grad re-listed in Figure 4.7. An outline of the corresponding ring computation in the implementation of the `computeGrad()` method, is shown in the code excerpt of Figure 4.8. Appendix A includes full code for all methods of the `CategoricalRegressionOverJoin` class.

Handling categorical features As noted in Section 4.3.1 above, the presence of categorical variables/features (and non-feature variables excluded from the linear regression model) requires complicated indexing and mapping to ensure that the correct elements of vectors of aggregates and parameters are used in the computations. For example, the code corresponding to the single highlighted computational step in Figure 4.7 is given in Figure 4.9, showing the use of the various indexes and mappings.

4.3.3 Supporting Methods

The 'core' methods to compute S_1, $S_{1,w}$ and Grad take both a factorised query join and d-tree as arguments. But they also rely on information about the linear regression model (the number of features; whether each feature is continuous or categorical or to be excluded completely from the model) and the set of values taken by any categorical query variables. This information is provided by supporting methods as follows (see also Figure 4.1):

- `loadFeatures()`: the d-tree is constructed from the dtree.txt file. This specifies, for each named query variable (data attribute) A a numeric identifier, its position in the d-tree – $anc(A)$ – and its dependencies (if any) on other query variables, that is, $key(A)$. Information about the linear regression model is stored in the variables.txt file: this specifies

```
// first pass per value and per child of cart product to compute sum of S1w values
double localS1w[numberOfValues];
std::fill_n(localS1w, numberOfValues, 0.0);
for (unsigned int val = 0; val < numberOfValues; ++val)
{
        DTreeNode* dTreeChild = dTreeNode->_firstChild;
        // sum S1w values
        for (int child = 0; child < numberOfChildren; ++child)
        {
                localS1w[val] += children[val * numberOfChildren + child]->S1w *
                                children[val * numberOfChildren + child]->multiplier;
                dTreeChild->_next;
        }
}
// second pass using local S1w values to compute local S values and update Grad
for (unsigned int val = 0; val < numberOfValues; ++val)
{
        DTreeNode* dTreeChild = dTreeNode->_firstChild;
        int index = 0;
        for (int child = 0; child < numberOfChildren; ++child)
        {
                // recursive call to computeGrad()
                computeGrad(children[val * numberOfChildren + child], dTreeChild);
                // calculate S for each child node using ring computation
                double localS = localS1w[val] -
                        children[val * numberOfChildren + child]->S1w *
                                children[val * numberOfChildren + child]->multiplier;
                // update Grad values
                for (int gradIndex = 0; gradIndex < dTreeChild->_numOfDescendentAggr;
                                                                        ++gradIndex)
                {
                        // Grad computation specific to each of conts/cat/non-features
                        dTreeChild = dTreeChild->_next;
                }
        }
}
```

Figure 4.8: Code excerpt from `computeGrad` method, showing ring computation (some details omitted for clarity - full code is provided in Appendix A).

```
if(_categoricalFeature[nodeID]){
       catAggrIndex = _catAggrToOverallAggrMapping[
                                           _nodeIDToCatAggrMapping[nodeID]][values[val]];
       paramIndexToUse = nodeInParamsIndex + attrInNodeIndex + catAggrIndex;
       factorisationNode->Grad[index] +=
              /* Grad * multiplicity */
       children[val * numberOfChildren + child]-> Grad[gradIndex] *
                     children[val * numberOfChildren + child]->multiplier +
              /* S*S1(j)[B]/cnt(j) */
       children[val * numberOfChildren + child]-> S1[gradIndex] * localS /
       children[val * numberOfChildren + child]->count
       // + additional terms from union part of algorithm
}
else if(_continuousFeature[nodeID])
{
       paramIndexToUse = nodeInParamsIndex + attrInNodeIndex;
       factorisationNode->Grad[index] +=
              /* Grad * multiplicity */
       children[val * numberOfChildren + child]-> Grad[gradIndex] *
              children[val * numberOfChildren + child]->multiplier +
              /* S*S1(j)[B]/cnt(j) */
       children[val * numberOfChildren + child]-> S1[gradIndex] * localS /
       children[val * numberOfChildren + child]->count
       // + additional terms from union part of algorithm
}
else if(_nonFeature[nodeID])
{
       paramIndexToUse = nodeInParamsIndex + attrInNodeIndex;
       factorisationNode->Grad[index] +=
              /* Grad * multiplicity */
       children[val * numberOfChildren + child]-> Grad[gradIndex] *
              children[val * numberOfChildren + child]-> multiplier +
              /* S*S1(j)[B]/cnt(j) */
       children[val * numberOfChildren + child]-> S1[gradIndex] * localS /
       children[val * numberOfChildren + child]->count;
}
```

Figure 4.9: Code excerpt from `computeGrad` method, corresponding to the highlighted computational step from the $\mathbf{F}^{\bigcirc}$ algorithm shown in Figure 4.7

whether each query variable corresponds to a continuous feature (=0), a categorical feature (=1), or neither (=2). The only constraint is that the response (= label) feature must correspond to a continuous query variable, since this is the feature predicted by the linear regression model. Any 'non-features' are disregarded for the purposes of gradient computation (though not, of course, for the construction of the query join). The `loadFeatures()` method in effect links the variable/attribute information for the query/d-tree with the feature information for the linear model. An existing implementation of `loadFeatures()` was used, with minor adaptations.

- `getCategoricalValues()`: for each categorical feature, this method invokes methods of the `CategoricalFactorisedJoinClass` to compute the set of values taken by the corresponding categorical variable in the query. This method, together with the four methods invoked on the `CategoricalFactorisedJoinClass` are new (see also Figure 4.5 above).

- `computeAggregateVectorInfo()`: as noted in the previous section, the set of possible values for each features determines the structure of the aggregate vectors at each stage in the recursive computation over the factorisation. Using the information returned by the `getCategoricalValues` method, this method populates the relevant instance variables for each DTreeNode object, together with the mappings between categorical values and their index positions in the aggregate vectors. This method is new.

- `_joinEngine->run()`: this invokes the methods of `CategoricalFactorisedJoin` to materialise the factorised representation of the natural join, using Leapfrog Triejoin. As noted above, an existing implementation of the `run()` method was reused to create the factorisation itself.

4.4 Implementation of Batch Gradient Descent

The main focus of the implementation is the rewritten ring computations; this is set within a simple batch gradient descent algorithm, implemented by the `optimize()` method from the existing codebase (outlined in Figure (4.6) above). This starts with an initial (randomly chosen) value for each w_j and updates at each convergence step according to the rule

$$w_j := w_j - \alpha \frac{\partial}{\partial w_j} J(w)$$

where each update uses a fresh computation of the gradients based on the current value of the parameters. The initial value for the learning rate α is set at 0.01; subsequent values are determined dynamically, based on the values of the current gradients. This is done using a standard technique known as the "adaptive" Barzilai-Borwein method, the properties of which are outlined by Goldstein et al. [8].

4.5 Unit Testing

The implementation was tested using small synthetic datasets with a range of combinations of continuous and categorical query variables (both with and without caching) to ensure that all factorised representations, aggregates, gradients, and model parameters were being computed correctly. In all cases, the relevant quantities were computed independently in SQLite (with pre one-hot encoding of categorical variables) and checked against the `C++` output from $\mathbf{F}^{\bigcirc}$.

Chapter 5

Experiments

Chapter 4 presented an implementation of $\mathbf{F}^{\bigcirc}$, a new algorithm exploiting ring computation for calculating query aggregates over the training dataset for a linear regression model. This chapter describes the experiments carried out on $\mathbf{F}^{\bigcirc}$. These enabled its performance to be compared against existing test data [16] for the industry state of the art (a development of the **F** system of Schleich et al, namely **DC** [16]), and with two open-source commercially available relational database systems, **MADlib** [12] 1.8 (**M**) and **R** [20] 3.0.2. Chapter 6 describes some current related academic work in the area of machine learning and databases; on the basis of the experimental outcomes below, Chapter 7 suggests directions for further work on the implementation of $\mathbf{F}^{\bigcirc}$.

5.1 Summary of Experimental Findings

Of the systems compared, $\mathbf{F}^{\bigcirc}$ is the only one able to handle the dataset size and number of categorical features in $\mathtt{v}_3$ and $\mathtt{v}_4$ in any reasonable timescale.

$\mathbf{F}^{\bigcirc}$ is 5x faster than the DC implementation of F algorithm for datasets v1 and v2 (in terms of the time to compute a complete set of aggregates required for a *single* convergence step of the batch gradient descent process). **DC** fails (insufficient memory) with the increased number of categorical features in $\mathtt{v}_3$ and $\mathtt{v}_4$. However, the benefits for **DC** of computing aggregates only once in the learning process more than compensate for the faster computation of those aggregates by $\mathbf{F}^{\bigcirc}$.

M is also unable to handle the higher number of categorical features in $\mathtt{v}_3$ and $\mathtt{v}_4$. The time that **M** takes to compute the model parameters analytically for $\mathtt{v}_1$ ($\mathtt{v}_2$) is equivalent to 374 (393) convergence steps of $\mathbf{F}^{\bigcirc}$'s (simple) batch gradient descent process. With a more sophisticated convergence process (see Section 7.2 below), it is likely that $\mathbf{F}^{\bigcirc}$ would outperform **M** on $\mathtt{v}_1$ and $\mathtt{v}_2$.

R fails for all except the smallest dataset with the fewest categorical features, $\mathtt{v}_1$. For the smallest dataset $\mathtt{v}_1$, **R** is able to compute the model features in less time than $\mathbf{F}^{\bigcirc}$ requires for its batch gradient process to converge, but $\mathbf{F}^{\bigcirc}$ is able to compute the query join more efficiently.

The most striking finding from the experimental timings for $\mathbf{F}^{\bigcirc}$, was that they effectively depend only on the size of the factorised dataset, and not on the number of parameters of the linear regression model. These timings (listed in Table 5.2) suggests that the main driver of complexity for $\mathbf{F}^{\bigcirc}$ in practice is the process of traversing the factorised join, rather than the computation carried out at each node of the factorisation. In order to benefit from the theoretical complexity improvements offered by $\mathbf{F}^{\bigcirc}$ it will therefore be necessary to reduce the number of traversals made by $\mathbf{F}^{\bigcirc}$ over the materialised factorisation and/or find more efficient data structures with which to represent the materialised join.

5.2 Experimental Approach

The key contribution of $\mathbf{F}^{\bigcirc}$ is the rewriting of the aggregates required to calculate a complete set of gradients for the least-squares objective function, that is for a single convergence step of the batch gradient descent process. The focus of the implementation for this project has been the implementation in `C++` of these rewritten aggregate computations, rather than the sophistication of the batch gradient descent process within which these computations are embedded. In particular, a standard adaptive learning rate α was used (the *adaptive BB method* [8]). Given the nature of the training data, in practice this resulted in frequent 'cycling' rather than strict convergence (in terms of the value of the least-squares objective function reducing at each step). To allow for per-convergence aggregate computation step timings to be compared, the number of convergence steps of the batch gradient descent process for $\mathbf{F}^{\bigcirc}$ was fixed at 200.

For **R** and **M** which do not use batch-gradient descent to solve the least-squares regression problem (rather, analytical models based on QR-decomposition and closed-form solutions respectively [16]) the comparison with $\mathbf{F}^{\bigcirc}$ is instead in terms of the number of batch gradient descent convergence steps that $\mathbf{F}^{\bigcirc}$ would be able to process in the time taken for **M** and **R** to compute the model parameters analytically, and in terms of some of the preprocessing tasks, such as the join time for **R**.

5.3 The Training Dataset

The training dataset used for the experiments was a real-life large-scale database for forecasting customer demand and sales for a large retailer. The database comprises five relations with the following schemas (with the **join variables** highlighted):

- Inventory(**locn**, **dateid**, **ksn**, inventoryunits). Records the inventory for each product at each store location on each date.
- Weather(**locn**, **dateid**, rain, snow, maxtemp, mintemp, meanwind, thunder). Records the weather at each store on each date.
- Census(**zip**, population, white, asian, pacific, black, medianage, + 9 more continuous variables). Records census (for customer demographic) information for each area by zip-code.
- Item(**ksn**, subcategory, category, categoryCluster, prize). Records information on the products sold at each store.
- Location(**locn**, **zip**, rgn_cd, clim_zn_nbr, tot_area_sq_ft, sell_area_sq_ft, + 8 more continuous variables). Records information about each of the retailer's stores.

The training dataset is the natural join of these five relations A snapshot of the (α-acyclic) d-tree for this natural join is shown in Figure 5.1. Where a query variable is treated as a categorical variable in one or more of the experiments, the number of distinct values taken by that variable in the query has been shown. Whilst necessary for the natural join of the five relations, **dateid** and **ksn** were excluded as model features in all experiments; in the case of **dateid**, this is because it would not make sense to make a linear projection on the basis of a cyclic feature such as the date. And incorporating the categorical variable stock number **ksn** would have resulted in over 6,400 additional model features, causing the size of the analytics to blow up.

Four different configurations of the training dataset were used, $\mathtt{v}_1$ to $\mathtt{v}_4$. The number of categorical query variables increases across the experiments, with a commensurate increase in the number of model features from 88 to 2691 as shown in Table 5.1. The size of the training dataset for $\mathtt{v}_1$ was smaller than that of $\mathtt{v}_2$ to $\mathtt{v}_4$: $\mathtt{v}_1$ was designed specifically to accommodate the limitations of **R** and $\mathtt{v}_1$/$\mathtt{v}_2$ the limitations of **M** [16] (see Section 5.4 below).

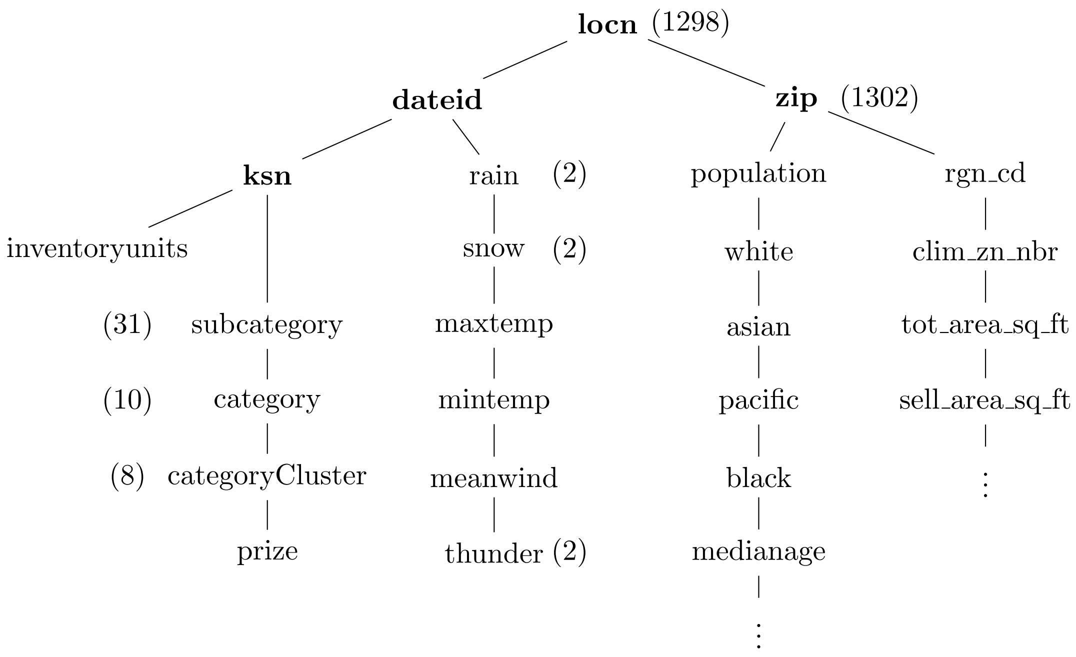


Figure 5.1: Snapshot of the Retailer d-tree (keys and root variable for intercept omitted) showing number of distinct values for variables that are categorical in one or more experiments.

	$\mathtt{v}_1$	$\mathtt{v}_2$	$\mathtt{v}_3$	$\mathtt{v}_4$
Flat join size of training dataset	7.74×10^8	3.61×10^9		
Factorised join size of training dataset	3.69×10^7	1.69×10^8		
Excluded features	dateid, ksn			
	locn, zip	locn, zip	locn	
Categorical features	subcategory, category cluster, rain, snow, thunder			
			zip	locn, zip
Continuous features	other 33 query variables			
Number of model features	88	88	1390	2691

Table 5.1: Configuration for experiments $\mathtt{v}_1$-$\mathtt{v}_4$, in terms of the size of training dataset used (that is, the number of values in flat/factorised representation of the query join) and the mix of continuous/ categorical/excluded model features.

5.4 Systems Tested Against

DC uses the **F** algorithm of Schleich et al. within a batch gradient descent process; however, it requires that any categorical features have already been one-hot encoded in the input data (with one column per categorical value, as in the one-hot encoded representations of relations $R_1(A, B, C)$ and $R_4(E, F)$ in Table 2.2(b)) [16]. This is in contrast to the implementation of $\mathbf{F}^{\bigcirc}$ which simply needs to know that a query variable *is* categorical; it then computes the appropriate number aggregates and learns the correct number of model parameters automatically. However, as with the original **F** concept [22], **DC** decouples the computation of database aggregates from the convergence steps of the batch gradient descent process; crucially, this means that the database aggregates need be calculated only once in the learning process. By contrast, $\mathbf{F}^{\bigcirc}$ requires a fresh set of aggregates to be calculated at each convergence step of the batch gradient descent process.

R [20] 3.0.2 is an open-source statistical analysis package; it was not compared against using its iterative batch gradient descent functionality, but instead its (faster) analytical *ols* approach to solving the least-squares minimisation problem. Limits on the number of values that **R** can process in its data frames means that the largest training dataset that it was able to handle was that of experiment $\texttt{v}_1$.

The other open-source system compared against was MADlib [12] 1.8 (**M**), also using an analytical rather than an iterative solution to the least-squares minimisation problem. **M** differs from both **F** and $\mathbf{F}^{\bigcirc}$ in how it handles categorical variables: it carries out its own one-hot encoding, requiring one column per categorical value; limits on the number of columns then constrain the number of categorical variables that can be accommodated [16].

5.5 Experimental Set-Up

To enable comparisons with existing experimental data for **DC**, **M**, and **R** [16], the machine used was an Intel(R) Core(TM) i7-4770 3.40GHz/64bit/32GB with Linux 3.13.0/g++4.8.4 (no compiler optimization flags were used, `ulimit` was set to unlimited). All tables were pre-sorted on their respective join attributes. This meant that the only sorting of relations required within $\mathbf{F}^{\bigcirc}$ was as part of the process to determine the set of categorical values for each categorical attribute (as noted in Section 4.2.2). Each of the four individual experiments $\texttt{v}_1$-$\texttt{v}_4$ was run five times. In each case, the time for the first run was discarded, and the average timings for each of the four remaining runs (that is, with a warm cache) reported. For $\mathbf{F}^{\bigcirc}$, the reported timings are broken down between individual code methods as follows:

(a) Creating the factorised join

```
_joinEngine->run();
```

(b) Pre-Processing Methods: each required to be performed only once, as the outputs of each method are independent of the values of the model parameters at any given convergence step of the batch gradient descent process

```
getCategoricalValues();
computeAggregateVectorInfo();
computeMultiplicity();
computeS1();
```

(c) First Convergence Step (including one computation of $S_{1,w}$ and Grad)

```
computeGradient() {
        computeS1w();
        computeGrad();
}
```

(d) Time for 200 Convergence Steps (each including one computation of $S_{1,w}$ and Grad)

```
computeGradient() {
        computeS1w();
        computeGrad();
}
```

(e) Average time for a single Convergence Step (over 200 steps).

As with the previous tests run on this dataset [16], the time to load the datasets into memory was not reported; the various systems perform very differently in this respect, and the focus for the testing was the computational performance of each system once the data had loaded.

	run 1	run 2	run 3	run 4	Avg
$\mathtt{v}_1$					
Pre-processing & first conv. step, o/w	**17.18**	**17.08**	**17.17**	**17.17**	**17.15**
construct factorised join	8.21	8.11	8.19	8.19	8.18
pre-processing (incl. Multiplicity/S_1)	3.93	3.93	3.93	3.93	3.93
first convergence step (incl. $S_{1,w}$/Grad)	5.04	5.04	5.04	5.04	5.04
Time for 200 convergence steps	**1006**	**1007**	**1007**	**1007**	**1007**
$\mathtt{v}_2$					
Pre-processing & first conv. step, o/w	**73.88**	**73.61**	**73.64**	**73.78**	**73.73**
construct factorised join	34.86	34.76	34.66	34.83	34.78
pre-processing (incl. Multiplicity/S_1)	16.53	16.59	16.53	16.50	16.49
first convergence step (incl. $S_{1,w}$/Grad)	22.49	22.46	22.45	22.46	22.46
Time for 200 convergence steps	**4488**	**4479**	**4479**	**4475**	**4480**
$\mathtt{v}_3$					
Pre-processing & first conv. step, o/w	**73.54**	**73.61**	**73.63**	**73.70**	**73.62**
construct factorised join	34.65	34.67	34.75	34.70	34.69
pre-processing (incl. Multiplicity/S_1)	16.42	16.47	16.43	16.50	16.46
first convergence step (incl. $S_{1,w}$/Grad)	22.47	22.47	22.45	22.50	22.47
Time for 200 convergence steps	**4483**	**4482**	**4478**	**4483**	**4482**
$\mathtt{v}_4$					
Pre-processing & first conv. step, o/w	**76.93**	**77.36**	**77.45**	**77.28**	**77.26**
construct factorised join	34.07	34.18	34.24	34.24	34.18
pre-processing (incl. Multiplicity/S_1)	20.38	20.64	20.60	20.46	20.52
first convergence step (incl. $S_{1,w}$/Grad)	22.48	22.54	22.61	22.57	22.55
Time for 200 convergence steps	**4485**	**4509**	**4513**	**4506**	**4504**

Table 5.2: Time performance comparison (seconds) for individual sets of $\mathbf{F}^{\bigcirc}$ methods when learning regression models over increasingly larger fragments ($\mathtt{v}_1$ to $\mathtt{v}_4$) of Retailer. All experiments were run for 200 convergence steps of the batch gradient descent process.

5.6 Breakdown of $\mathbf{F}^{\bigcirc}$ Results

The breakdown of individual execution timings for the various $\mathbf{F}^{\bigcirc}$ methods are given in Table 5.2. The striking (and unexpected) feature of the results (repeated over several independent runs of the experiments) is that they seem largely independent of the size of the associated linear regression model for a given size of database. With the rewritten Grad calculations, the computation time for $S_{1,w}$/Grad is certainly not quadratic in the size of the model; it would be expected to be linear, but in fact is close to constant! A further breakdown of the $\mathbf{F}^{\bigcirc}$ results shows that for each experiment, the times to compute a set of gradients (including $S_{1,w}$ and Grad) are actually very close to double the times to compute S_1, as shown in Table 5.3 for the four datasets. Given that each of the three methods `computeS1()` `computeS1w()` and `computeGrad()` involves a single pass over the factorised dataset, this suggests that the dominant factor in the complexity is in fact the time to traverse the factorisation, rather than the time to perform the actual calculations at each node. This may be a consequence of the fact that the representation of the factorisation uses linked nodes, which makes for costly operations to find and process child and sibling nodes within the recursive computations. The difference between the pre-processing times for $\mathtt{v}_3$ and $\mathtt{v}_4$ is due to the additional work to obtain an extra 1300 categorical values, in particular the sorting required on each of the zip and locn values individually (as noted in Section 4.2.2). And the reduced times for $\mathtt{v}_1$ reflects the fact that the dataset is one-fifth of the size of the dataset for $\mathtt{v}_2$ to $\mathtt{v}_4$.

	run 1	run 2	run 3	run 4	Avg
$\mathtt{v}_1$					
`computeS1()`	2.52	2.53	2.53	2.53	2.53
`computeGradient()` (incl. $S_{1,w}$/Grad)	5.04	5.04	5.04	5.04	5.04
$\mathtt{v}_2$					
`computeS1()`	11.19	11.12	11.22	11.15	11.17
`computeGradient()` (incl. $S_{1,w}$/Grad)	22.49	22.46	22.45	22.46	22.46
$\mathtt{v}_3$					
`computeS1()`	11.14	11.17	11.15	11.23	11.17
`computeGradient()` (incl. $S_{1,w}$/Grad)	22.47	22.47	22.45	22.50	22.47
$\mathtt{v}_4$					
`computeS1()`	11.09	11.18	11.19	11.11	11.14
`computeGradient()` (incl. $S_{1,w}$/Grad)	22.48	22.54	22.61	22.57	22.55

Table 5.3: Time performance comparison (seconds) for the $\mathbf{F}^{\bigcirc}$ methods `computeS1()`, and `computeGradient()` over Retailer datasets $\mathtt{v}_1$ to $\mathtt{v}_4$.

5.7 Results of $\mathbf{F}^{\bigcirc}$ Against Other Systems

A comparison of the performance of $\mathbf{F}^{\bigcirc}$ against **DC**, **M** and **R** is given in Table 5.4. Some points to note:

Given that the time for $\mathbf{F}^{\bigcirc}$ to compute a first set of gradients includes the time to compute the aggregates required for a set of gradients, it can be seen that $\mathbf{F}^{\bigcirc}$ offers a small performance increase over **DC** for computing a single set of aggregates. However, as noted in Chapter 3, a disadvantage of $\mathbf{F}^{\bigcirc}$ is that it re-couples the computation of aggregates with the computation of the gradients at each convergence step of the batch gradient descent process. Unless convergence occurs within 15 convergence steps of the batch gradient descent process for $\mathbf{F}^{\bigcirc}$ (this is vanishingly unlikely, given the current adaptive learning rate used by $\mathbf{F}^{\bigcirc}$) **DC** will still be faster overall. That said, the combined memory requirements of a decoupled approach using the Cofactor matrix and one-hot pre-encoding of categorical variables are simply too high for **DC** to be able to process the training datasets for experiments $\mathtt{v}_3$ and $\mathtt{v}_4$. $\mathbf{F}^{\bigcirc}$, on the other hand, is able to process the increasing numbers of categorical variables with only a very marginal increase in times over that for $\mathtt{v}_2$.

$\mathbf{F}^{\bigcirc}$ outperforms **R** in terms of the time taken to compute the join (a factorised join in the case of $\mathbf{F}^{\bigcirc}$ and a flat join in the case of **R**. Whilst **R**'s analytic solution to the least-squares minimisation problem is certainly faster than $\mathbf{F}^{\bigcirc}$ with its current adaptive learning rate, **R** is unable to compute parameters for any training dataset larger than that of $\mathtt{v}_1$. And with a more efficient learning rate, it is highly likely (and almost certain in the case of $\mathtt{v}_3$) that $\mathbf{F}^{\bigcirc}$ would outperform **M**.

		$\mathtt{v}_1$	$\mathtt{v}_2$	$\mathtt{v}_3$	$\mathtt{v}_4$
Feats	(conts + cat)	33 + 55	33+55	33+1340	33+2658
	Aggregates	595+2,418	595+2,421	595+111,549	595+157,735
M	Learn (ols)	1,898.35	8,855.11	$> 79,200.00$	–
R	Join (PSQL)	50.63	–	–	–
	Export/Import	308.83	–	–	–
	Learn (qr)	490.13	–	–	–
$\mathbf{F}^{\bigcirc}$	**Time to first conv. step**	**17.15**	**73.73**	**73.62**	**77.26**
o/w	Join	8.18	34.78	34.69	34.18
	Pre-proc	3.93	16.49	16.46	20.52
	First conv. step	5.04	22.46	22.47	22.55
	Time 200 conv. steps	**1007**	**4480**	**4482**	**4504**
	Avg time/step	**5.02**	**22.32**	**22.33**	**22.44**
DC	Aggregate	93.31	424.81	OOM	OOM
	Converge (runs)	0.01 (359)	0.01 (359)		
Speedup for computation of first set of aggregates					
	$\mathbf{F}^{\bigcirc}$**/DC**	5×	5×	∞	∞
Upper limit on $\mathbf{F}^{\bigcirc}$ bgd iterations for $\mathbf{F}^{\bigcirc}$ to outperform competitors					
	$\mathbf{F}^{\bigcirc}$**/DC**	15	15	∞	∞
	$\mathbf{F}^{\bigcirc}$**/M**	374	393	> 3500	∞
	$\mathbf{F}^{\bigcirc}$**/R**	165	∞	∞	∞

Table 5.4: Time performance comparison (seconds) for learning a linear regression model over increasingly larger fragments ($\mathtt{v}_1$ to $\mathtt{v}_4$) of Retailer. The join size stays the same for versions $\mathtt{v}_2$ to $\mathtt{v}_4$ at 3,614,400,131 values as flat vs 169,231,200 values as factorized (21.4× compression); $\mathtt{v}_1$ is a fifth of $\mathtt{v}_2$, with 774M values as flat vs 36,929,272 values as factorized (20.96× compression). (–) means that the system failed to compute due to design limitations. R can only compute the model for $\mathtt{v}_1$, the other versions exceed the size limit of R's data frames. M cannot compute any model on $\mathtt{v}_4$ since the one-hot encoding requires more than 1600 columns. **M/R/DC** data from [16]; $\mathbf{F}^{\bigcirc}$ data is new.

Chapter 6

Related Work

Previous chapters of this dissertation have considered the state of the art for learning the parameters of linear regression models over the materialised factorisation of the result of any join query, including an implementation of the new $\mathbf{F}^{\bigcirc}$ algorithm in Chapter 4 and its empirical performance in Chapter 5. This chapter describes briefly some related academic work in the crossover between machine learning and databases. Chapter 7 then summarises the outcomes of the current project on which this dissertation is based, and suggests directions for further work to improve the implementation of $\mathbf{F}^{\bigcirc}$.

6.1 In-database Machine Learning

Artificial Intelligence, including machine learning, is a commercially significant, and expanding, area: The Economist estimated that, in 2015, technology companies invested \$8.5 billion on deals and investments in artificial intelligence, a fourfold increase on 2010, and a far cry from a field that had been "largely ignored and underfunded" in previous decades [2]. In May 2015, a panel discussion at the ACM Special Interest Group on Management of Data (SIGMOD) acknowledged the "new breed of high-value data-driven applications in image analysis, search, voice recognition, mobile and office productivity products" and "the increasing crossover between database research and machine learning", and asked what the role and contribution of the academic database community to this trend should be. For example, was there anything fundamentally different about building database systems for machine learning, and were there long-standing database problems where machine learning could help provide a solution [21].

Insofar as the work in this dissertation considers the situation where the input to a machine learning problem is given by a relation representing the result of a join query over a relational database, it lies precisely within this crossover! It builds on the work on factorised databases, pioneered by Olteanu and colleagues [4][5][17][18][19][22], in particular the insight that by removing the redundancy in the listing representation of queries (which is not required for machine learning over the data) it is possible to speed up the computation and analytics on the data [18].

Going beyond linear regression models, the use of factorised datasets to learn regression models can be applied to *any* regression model where the gradients of the objective function can be expressed in terms of the semiring operations on which the factorisations rely [18]. Recent work in this area by Ngo et al. has extended the class of regression models to include includes Ridge Linear Regression (mentioned in Section 2.2.3 above), polynomial regression, factorization machines, classification, and principal component analysis [16]. Their work provided a practical answer to the cultural challenge posed by the SIGMOD panel, in terms of why would in-database solutions be any more efficient than the specialist analytical tools and techniques currently available for out-of-database processing. In fact it demonstrated that, using specifically 'database-centric' technical contributions, a model can be trained in time sub-linear in the output size of the feature extraction query. These technical contributions included: more efficient approaches to the one-hot encoding of categorical variables; removing redundancy from the learning process by excluding from the process that are functionally dependent on other model features; and exploiting join dependencies in the training dataset to improve the computation times of the gradient descent process. Directions for future research identified by Ngo et al. include: widening the class of statistical models for in-database training; adapting other optimization algorithms than batch gradient descent (for example coordinate descent or stochastic gradient); and further exploration of the impact of functional dependency based reductions in the dimensions of the regression model.

Factorised databases also have parallels with the Functional Aggregate Query (FAQ) problem [13] which can be thought of as a declarative query language over functions. The input to the FAQ problem is a set of input functions (or factors) over a set of variables, and the output is calculated by aggregating over the variables and input factors, where the aggregating functions form a semi-ring. FAQ can thus be viewed as a generalisation of factorised databases (FDB): the FAQ framework shows how the FDB problem and computation algorithm can be applied to many other applications (including matrix operations, logic, and constraint satisfaction problems) simply by changing the semi-ring operations. The technical contributions in the work of Ngo et al. discussed here make use of FAQ expressions in the handling of one-hot encoded features.

Chapter 7

Conclusions and Future Work

This chapter summarises the outcomes of the current project on which this dissertation is based, and suggests directions for further work to improve the implementation of $\mathbf{F}^{\bigcirc}$.

7.1 Conclusions

This dissertation introduced a new algorithm $\mathbf{F}^{\bigcirc}$ to support the efficient supervised learning of linear regression model parameters over arbitrary natural joins of database relations. The algorithm accommodates arbitrary combinations of continuous and categorical query variables (its implementation also gives the flexibility for any query variable to be excluded completely as a feature of the linear regression model). Moreover, the algorithm has also been shown to offer asymptotic complexity improvements over the state of the art in this area, namely the **F** algorithm of Schleich et al. [22], especially in the presence of categorical query variables.

The algorithm was implemented in `C++` and was successfully unit-tested against small synthetic datasets with a range of combinations of continuous and categorical query variables (including the use of caching) to ensure that the factorised representations, aggregates, and gradients were being computed correctly.

The performance of $\mathbf{F}^{\bigcirc}$ was investigated using four large-scale real-world retailer datasets of increasing size and with greater numbers of categorical data quantities. The outcomes were compared with existing test data using the same datasets [16] for the industry state of the art (a development of the **F** system of Schleich et al, namely **DC** [16]), and with two open-source commercially available relational database systems, **MADlib** [12] 1.8 (**M**) and **R** [20] 3.0.2. Out of the four systems compared, only $\mathbf{F}^{\bigcirc}$ was able to compute model parameters for the larger datasets with greater numbers of categorical variables; on the smaller datasets $\mathbf{F}^{\bigcirc}$ was also 5x faster than the implementation of **F** in terms of computing a single set of database aggregates.

Notwithstanding the success of $\mathbf{F}^{\bigcirc}$ in handling large numbers of categorical values, the tests on the retailer datasets did not, however, demonstrate in practice the expected asymptotic complexity improvements from the use of ring computation. Instead, computation times appeared to be dominated by the data size of the query, and were constant in the size of the linear regression model (rather than linear as in the complexity analysis of the $\mathbf{F}^{\bigcirc}$ algorithm). In particular the faster times to compute a single set of aggregates for $\mathbf{F}^{\bigcirc}$ compared with **F** were not sufficient to outweigh the benefits that **F** gains by calculating database aggregates only once in the end-to-end learning process. Suggestions for further work to improve the implementation of $\mathbf{F}^{\bigcirc}$ are set out in the following section.

7.2 Future Work

The current implementation of $\mathbf{F}^{\bigcirc}$ has two key areas for further work and improvement:

Convergence of the Batch Gradient Descent Process A consequence of $\mathbf{F}^{\bigcirc}$ needing to recompute the full set of query aggregates at each step of the batch gradient descent process is that any inefficiency in the convergence of that process will result in steep time-performance penalties. The current implementation uses an adaptive (Barzilai-Borwein) learning rate. In practice, however, this is not sufficient to ensure that each successive step of the batch gradient descent process results in a lower value of the least-squares objective function being minimised (demonstrated by the fact that the ℓ_2 norm of the gradients sometimes increases rather than decreases from one iterative convergence step to the next). Intuitively, this is an indication that the step-size at that point is too large, thus overshooting the minimum that the process is trying to find. The first area for further work is therefore to investigate the use of more sophisticated convergence techniques, such as *backtracking line search*; this limits the learning rate by checking at each iteration that the objective function has decreased sufficiently and, if not, *backtracking* until this is the case [8].

Data Structures Used for the Factorised Join As noted above, the experimental results from the large-scale datasets did not reflect the theoretical performance gains expected from the rewritten computations in the $\mathbf{F}^{\bigcirc}$ algorithm: the times for the computation of each set of aggregates were dependent on the size of the factorised query (as expected), but independent of the size of the linear regression model. This suggests that the dominant factor in the practical execution time is the work to navigate in memory *between* the nodes of the factorisation, rather than to perform the computations *at* each node of the factorisation. As noted in Chapter 4, the factorisation of the query join is materialised in memory using a tree of linked `Union` structs; the second area for further work is therefore to reduce the number of passes over the factorisation per set of aggregates, and to investigate alternative, more efficient data structures for the factorisation itself.

Appendix A

CategoricalRegressionOverJoin.cpp

This Appendix contains the full code of the CategoricalRegressionOverJoin class. The structure of the class was modelled on an existing class for another regression method (RegressionOverJoin), but extended with new/modified methods. Comments before each method indicate the extent to which existing code has been reused.

```
/*
 * CategoricalRegressionOverJoin.cpp
 *
 *  Created on: Jul 5, 2017 - version for experiments
 *      Author: ruth wells
 *
 *  High level structure based on RegressionOverJoin.cpp
 *  but with new/modified methods.
 */

#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/spirit/include/qi.hpp>
#include <cstdlib>
#include <fstream>
#include <cmath>
#include <ctime>
#include <GlobalParams.hpp>
#include <Launcher.h>
#include <Logging.hpp>

#include <CategoricalRegressionOverJoin.h>

static const std::string FEATURE_CONF = "/variables.txt";

static const char COMMENT_CHAR = ’#’;
static const char NUMBER_SEPARATOR_CHAR = ’,’;
static const char PARAMETER_SEPARATOR_CHAR = ’ ’;
static const char ATTRIBUTE_SEPARATOR_CHAR = ’,’;
static const char ATTRIBUTE_NAME_CHAR = ’:’;

namespace phoenix = boost::phoenix;
using namespace boost::spirit;
using namespace std;
using namespace std::chrono;
using namespace dfdb::params;
using namespace dfdb::types;

/*
 *  Constructor and destructor - reused from RegressionOverJoin
 */
```

```
CategoricalRegressionOverJoin::CategoricalRegressionOverJoin(const std::string&
               pathToFiles, std::shared_ptr<Launcher> launcher) :
                              _pathToFiles(pathToFiles), _launcher(launcher)
{
       _dTree = launcher->getDTree();
};

CategoricalRegressionOverJoin::~CategoricalRegressionOverJoin(){};

/*
 * CategoricalRegressionOverJoin::run()
 *
 * Modified from RegressionOverJoin::run to incorporate new methods for handling
 * categorical variables.
 *
 * Computes a CategoricalRegression model
 */
void CategoricalRegressionOverJoin::run()
{
       loadFeatures();

       _joinEngine = (std::dynamic_pointer_cast<CategoricalFactorisedJoin>
                                            (_launcher->getAggregateEngine()));

#ifdef BENCH
       int64_t startGetCatValues = duration_cast<milliseconds>(
                      system_clock::now().time_since_epoch()).count();
#endif

       /* Get all our categorical values here */
       getCategoricalValues();
       /* and also construct the info for the aggregate vectors */
       DTreeNode* dTreeRoot = _dTree->_root;
       computeAggregateVectorInfo(dTreeRoot);

#ifdef BENCH
    int64_t timeCatValues = duration_cast<milliseconds>
           (system_clock::now().time_since_epoch()).count() - startGetCatValues;
    /* Write Categorical Values Computation times to times file */
    std::ofstream ofs("times.txt", std::ofstream::out | std::ofstream::app);
    if (ofs.is_open())
       ofs << "getCatValues: \t"+to_string(timeCatValues) + "\n";
    else
       cout << "Unable to open times.txt \n";
    ofs.close();
#endif

BINFO(
       "WORKER - categorical values and aggregate vector info: " +
                                    std::to_string(timeCatValues) + "ms.\n");
DINFO(
       "WORKER - categorical values and aggregate vector info have
                                    been determined. \n");

#ifdef BENCH
       int64_t startJoin = duration_cast<milliseconds>(
                      system_clock::now().time_since_epoch()).count();
#endif

       /* create factorisation */
       _joinEngine->run();
```

```
        rootNode = _joinEngine -> getFactorisationRoot();

#ifdef BENCH
    int64_t timeJoin = duration_cast<milliseconds>
           (system_clock::now().time_since_epoch()).count() - startJoin;
    /* Write factorisation times to times file */
    ofs.open("times.txt", std::ofstream::out | std::ofstream::app);
    if (ofs.is_open())
        ofs << "joinConstruction: \t"+to_string(timeJoin) + "\n";
    else
        cout << "Unable to open times.txt \n";
    ofs.close();
#endif

BINFO(
        "WORKER - join construction: " + std::to_string(timeJoin) + "ms.\n");

DINFO(
        "WORKER - join has been constructed. \n");

#ifdef BENCH
        int64_t start = duration_cast<milliseconds>(
                      system_clock::now().time_since_epoch()).count();
#endif

        /* compute parameters (right through to optimize()) */
        computeParameters();

#ifdef BENCH
    int64_t timeComputeParameters = duration_cast<milliseconds>
           (system_clock::now().time_since_epoch()).count() - start;
    /* Write overall parameter computation times to times file */
    ofs.open("times.txt", std::ofstream::out | std::ofstream::app);
    if (ofs.is_open())
        ofs << "convergence - total: \t"+to_string(timeComputeParameters) + "\n" +
               "--------------------------------------------------------- \n";
    else
        cout << "Unable to open times.txt \n";
    ofs.close();
#endif

BINFO(
        "MASTER - convergence - total: "
                      + to_string(timeComputeParameters) + "ms.\n");

BINFO("Final gradients [");
for (size_t attr = 0; attr < NUM_OF_PARAMETERS; ++attr)
{
        DINFO("gradient["+to_string(attr)+"] = " +
                             to_string(_gradient[attr]) + "\n");
        BINFO(" "+to_string(_gradient[attr]));
}
BINFO("]\n");

BINFO("Final parameters [");
for (size_t param = 0; param < NUM_OF_PARAMETERS; ++param)
{
        BINFO(" "+to_string(_parameters[param]));
}
BINFO("]\n");
```

```
        exit(0);
};

/*
 * CategoricalRegressionOverJoin::computeParameters()
 *
 * Modified significantly from RegressionOverJoin::computeParameters() to
 * incorporate new methods for handling categorical variables and to create/
 * populate the various indices and mappings for the vectors of aggregates.
 *
 * Learns a set of regression model parameters
 */
void CategoricalRegressionOverJoin::computeParameters()
{
        int dTreeRootId = _dTree->_root->_id;
        unsigned int numberOfCachedValues =
                        _joinEngine->getNumberOfCachedValues();

        /* create the caches */
        if (numberOfCachedValues > 0)
        {
                _S1Cached = new bool[numberOfCachedValues]();
                _S1wCached = new bool[numberOfCachedValues]();
                _GradCached = new bool[numberOfCachedValues]();
        }

        /* Initialise the parameters to random values. */

        _parameters = new double[NUM_OF_PARAMETERS];
        /* seed the random number generator */
        srand(time(NULL));
        for (size_t parameter = 0; parameter < NUM_OF_PARAMETERS; ++parameter)
                _parameters[parameter] =
                        ((ResultType) (rand() % 200 + 1) - 100) / pow(10.0, 10);

        /* fix the parameter at -1 for the label/response feature */
        LABEL_PARAMETER_INDEX = _dTree->_root->
                        _indexOfAttrInNodeAggregates[labelID];
        _parameters[LABEL_PARAMETER_INDEX] = -1;
        /* and we don't want parameters for non-features */
        for (int attrID = 0; attrID < NUM_OF_ATTRIBUTES; ++ attrID)
        {
                if (_nonFeature[attrID])
                {
                        int attrParamIndex = _dTree->_root->
                                _indexOfAttrInNodeAggregates[attrID];;
                        _parameters[attrParamIndex] = 0;
                }
        }

        _previousParameters = new ResultType[NUM_OF_PARAMETERS];
        for (size_t parameter = 0; parameter < NUM_OF_PARAMETERS; ++parameter)
        {
                _previousParameters[parameter] = _parameters[parameter];
        }

        _gradient = new ResultType[NUM_OF_PARAMETERS]();
        _previousGradients = new ResultType[NUM_OF_PARAMETERS]();

        /* code to populate the various indexes that link attributes
           with model parameters and query variables/attributes */
```

```
       for (int attrID = 0; attrID < NUM_OF_ATTRIBUTES; ++attrID)
       {
              DTreeNode* currentNode = _dTree->getNode(attrID);
              /* retrieve the first attribute in the traversal of this node */
              int firstAttr = currentNode->_traversal[0];
              /* find the corresponding position in the overall d-tree */
              int firstAttrPos = _dTree->_root->
                            _indexOfAttrInNodeAggregates[firstAttr];
              /* store this in the current DTreeNode */
              currentNode->_indexOfNodeInRootAggregates = firstAttrPos;
       }

#ifdef BENCH
       int64_t startMultiplicity = duration_cast<milliseconds>(
                     system_clock::now().time_since_epoch()).count();
#endif
       /* compute multiplicities */
       computeMultiplicity(rootNode, _dTree->_root, 1);

#ifdef BENCH
    int64_t timeMultiplicity = duration_cast<milliseconds>
           (system_clock::now().time_since_epoch()).count() - startMultiplicity;
    /* Write Multiplicity Computation times to times file */
    std::ofstream ofs("times.txt", std::ofstream::out | std::ofstream::app);
    if (ofs.is_open())
       ofs << "convergence: multiplicity: \t"+to_string(timeMultiplicity) + "\n";
    else
       cout << "Unable to open times.txt \n";
    ofs.close();
#endif

    BINFO(
"MASTER - convergence: multiplicity: " + to_string(timeMultiplicity) + "ms.\n");

#ifdef BENCH
       int64_t startS1 = duration_cast<milliseconds>(
                     system_clock::now().time_since_epoch()).count();
#endif

       /* compute S1 aggregates */
       computeS1(rootNode, _dTree->_root);

#ifdef BENCH
    int64_t timeS1 = duration_cast<milliseconds>
           (system_clock::now().time_since_epoch()).count() - startS1;
    /* Write S1 Computation times to times file */
    ofs.open("times.txt", std::ofstream::out | std::ofstream::app);
    if (ofs.is_open())
       ofs << "convergence: sums: \t"+to_string(timeS1) + "\n";
    else
       cout << "Unable to open times.txt \n";
    ofs.close();
#endif

DINFO("Sums Computed! \n");

BINFO(
"MASTER - convergence: sums: " + to_string(timeS1) + "ms.\n");

#ifdef BENCH
```

```
       int64_t startOptimize = duration_cast<milliseconds>(
                      system_clock::now().time_since_epoch()).count();
#endif

       /* run the batch gradient descent iterations */
       optimize();

#ifdef BENCH
    int64_t timeOptimize = duration_cast<milliseconds>
           (system_clock::now().time_since_epoch()).count() - startOptimize;
    /* Write Optimize Computation times to times file */
    ofs.open("times.txt", std::ofstream::out | std::ofstream::app);
    if (ofs.is_open())
       ofs << "convergence: optimize - total: \t"+to_string(timeOptimize) + "\n";
    else
       cout << "Unable to open times.txt \n";
    ofs.close();
#endif

DINFO("NUM_OF_PARAMETERS: "+to_string(NUM_OF_PARAMETERS) + "\n");
BINFO(
"MASTER - convergence: optimize - total: " + to_string(timeOptimize) + "ms.\n"+
"MASTER - number of iterations while producing final convergence result: "
                      + to_string(_numOfIterations) + "\n");

DINFO("\n");
       for (size_t i = 0; i < NUM_OF_PARAMETERS; ++i)
              DINFO("param["+to_string(i)+"] : "+to_string(_parameters[i])+" \n");
};

/*
 * CategoricalRegressionOverJoin::loadFeatures()
 *
 * Modified from RegressionOverJoin::loadFeatures to incorporate
 * categorical variables.
 *
 * Loads the text file variables.txt to link d-tree information on attributes
 * with linear regression model information on features.
 */

void CategoricalRegressionOverJoin::loadFeatures()
{
       /* set up arrays to hold info on type of each feature */
       _continuousFeature = new bool[NUM_OF_ATTRIBUTES]();
       _categoricalFeature = new bool[NUM_OF_ATTRIBUTES]();
       _nonFeature = new bool[NUM_OF_ATTRIBUTES]();

       /* Load the two-pass variables config file into an input stream. */
       ifstream input(_pathToFiles + FEATURE_CONF);
       if (!input)
       {
              ERROR(_pathToFiles + FEATURE_CONF+" does not exist. \n");
              exit(1);
       }

       /* String and associated stream to receive lines from the file. */
       string line;
       stringstream ssLine;
       int numOfLinearFeatures = 0;
       int degreeOfInteractions = 0;
       int numOfManualInteractions = 0;
```

```
/* Ignore comment and empty lines at the top */
while (getline(input, line))
{
       if (line[0] == COMMENT_CHAR || line == "")
              continue;
       break;
}

/*
* Parse the first line with the three numbers; ignore spaces.*/
bool parsingSuccess =
       qi::phrase_parse(line.begin(), line.end(),
/* Begin Boost Spirit grammar. */
       ((qi::int_[phoenix::ref(numOfLinearFeatures) = qi::_1])
              >> NUMBER_SEPARATOR_CHAR
              >> (qi::int_[phoenix::ref(degreeOfInteractions) =
                            qi::_1]) >> NUMBER_SEPARATOR_CHAR
              >> (qi::int_[phoenix::ref(numOfManualInteractions) =
                                                         qi::_1])),
/*  End grammar. */

ascii::space);

assert(parsingSuccess && "The parsing of the features file has failed.");
if (degreeOfInteractions > 3)
{
       ERROR("degree more than 3 is currently not supported. ");
       exit(1);
}
if (degreeOfInteractions < 2)
       degreeOfInteractions = 1;

#ifdef DEGREE
       if(DEGREE != degreeOfInteractions)
       {
              ERROR("Value of compiler flag -DDEGREE and degree
of interactions specified in features.txt inconsistent. Aborting.\n")
              exit(1);
       }
       /* Assign value to extern variable in GlobalParams.hpp. */
#else
       DEGREE_OF_INTERACTIONS = degreeOfInteractions;
#endif

#ifdef MANUAL
       if(MANUAL != numOfManualInteractions)
       {
              ERROR("Value of compiler flag -DMANUAL and number of manual
interactions specified in features.txt inconsistent. Aborting.\n")
              exit(1);
       }
       /* Assign value to extern variable in GlobalParams.hpp. */
#else
       NUM_OF_MANUAL_INTERACTIONS = numOfManualInteractions;
#endif

std::vector<int> linearContinuousFeatures;

int numOfCategoricalFeatures = 0;
int numOfContinuousFeatures = 0;
int numOfNonFeatures = 0;

/* Read in the features. */
for (int featureNo = 0; featureNo < numOfLinearFeatures; ++featureNo)
{
       getline(input, line);
       if (line[0] == COMMENT_CHAR || line == "")
       {
              --featureNo;
              continue;
       }

       ssLine << line;
       string attrName;
       /* Extract the name of the attribute in the current line. */
       getline(ssLine, attrName, ATTRIBUTE_NAME_CHAR);
       string typeOfFeature;
       /* Extract the dimension of the current attribute. */
       getline(ssLine, typeOfFeature);

       int attributeID = _dTree->getIndexByName(attrName);
       int typeoffeature = stoi(typeOfFeature);

       if (attributeID == -1)
       {
              ERROR("Attribute |"+attrName+"| does not exist.");
              exit(1);
       }

       /* check that label feature is continuous! */
       if (featureNo == 0 && typeoffeature != 0)
       {
              ERROR("The label needs to be continuous! ");
              exit(1);
       }

       if (featureNo == 0)
              labelID = attributeID;

       /* determine the type of each feature: conts, cat, or non-feature */
       switch (typeoffeature)
       {
              case 0 :
                     _continuousFeature[attributeID] = true;
                     ++numOfContinuousFeatures;
                     break;
              case 1 :
                     _categoricalFeature[attributeID] = true;
                     ++numOfCategoricalFeatures;
                     break;
              case 2:
                     _nonFeature[attributeID] = true;
                     ++numOfNonFeatures;
       }
       /* Clear string stream. */
       ssLine.clear();
}

if (numOfLinearFeatures + numOfManualInteractions == 0)
       ERROR("There should be at least one feature in the dataset \n");
```

```
        #ifdef LFEATURES
                if(LFEATURES != numOfContinuousFeatures)
                {
                        ERROR("Value of compiler flag -DLINEAR and number of linear
                features specified in features.txt inconsistent. Aborting.\n")
                        exit(1);
                }
                /* Assign value to extern variable in GlobalParams.hpp. */
        #else
                NUM_OF_LINEAR_FEATURES = numOfContinuousFeatures;
        #endif

        int numOfFeatures = numOfContinuousFeatures + 1;

        /* Check consistency between compile time flag and runtime value
                        issued from configuration file. */
        #ifdef FEATURES
                if(FEATURES != numOfFeatures)
                {
                        ERROR("Value of compiler flag -DFEATURES and number
                of features specified in features.txt inconsistent. Aborting.\n")
                        exit(1);
                }
                /* Assign value to extern variable in GlobalParams.hpp. */
        #else
                NUM_OF_FEATURES = numOfFeatures;
        #endif

        /* record the number of categorical features and non-features */
        NUM_OF_CAT_FEATURES = numOfCategoricalFeatures;
        NUM_OF_NON_FEATURES = numOfNonFeatures;
};


/*
 * CategoricalRegressionOverJoin::getCategoricalValues()
 *
 * New method
 *
 * Determines the complete set of values taken by any categorical
 * query variable.
 */
void CategoricalRegressionOverJoin::getCategoricalValues()
{
        /* set up the map we will need for sums and gradients */
        /* at this point NUM_OF_CAT_VARIABLES = number of cat attrs! */
        _catAggrToOverallAggrMapping =
                new std::unordered_map<double, int>[NUM_OF_CAT_FEATURES];
        // TODO - need to think about how to delete this idc!!
        int numberCategoricalFeatures = 0;
        int catVariableNo = 0;
        for (size_t attrID = 0; attrID < NUM_OF_ATTRIBUTES; ++attrID)
        {
                if (_categoricalFeature[attrID])
                {
                        _joinEngine->
                sortDataForCategoricalValues(_dTree->getNode(attrID));
                        categoricalValues = _joinEngine->
                                getCategoricalValues(_dTree->getNode(attrID));
                        unsigned int numCatValues =
                                static_cast<unsigned int>(categoricalValues[1]+0.1);
                        numberCategoricalFeatures += numCatValues;

                        /* populate our maps */
                        _nodeIDToCatAggrMapping.insert({attrID, catVariableNo});
                        for (int i = 0; i < numCatValues; ++i)
                                _catAggrToOverallAggrMapping[catVariableNo].insert
                                                ({categoricalValues[i+2], i});
                        ++catVariableNo;

                        /* record how many categoricalAggregates for this
                         * dTreeNode */
                        _dTree->getNode(attrID)->_numCatValues = numCatValues;
                }
        }

        NUM_OF_CAT_FEATURES = numberCategoricalFeatures;
        NUM_OF_PARAMETERS = NUM_OF_FEATURES +
                                NUM_OF_CAT_FEATURES + NUM_OF_NON_FEATURES;
};

/*
 * CategoricalRegressionOverJoin::computeAggregateVectorInfo()
 *
 * New method
 *
 * Determines the structure of the vector of aggregates for each Union struct
 * in the factorisation, based on the structure of the d-tree and the values
 * taken by any categorical variables, creating the required indexes and
 * mappings.
 */
void CategoricalRegressionOverJoin::
                                computeAggregateVectorInfo(DTreeNode* dTreeNode)
{
        int numberOfChildren = dTreeNode->_numOfChildren;
        int nodeID = dTreeNode->_id;

        if (numberOfChildren == 0)
        {
                dTreeNode->_traversal.push_back(nodeID);
                dTreeNode->_indexOfAttrInNodeAggregates.insert({nodeID, 0});
                if (_categoricalFeature[nodeID])
                {
                        dTreeNode->_numOfDescendentAggr =
                                                        dTreeNode->_numCatValues;
                }
                else // conts or non-feature
                {
                        dTreeNode->_numOfDescendentAggr = 1;
                }
        }
        else
        {
                int index = 0;
                int sizeOfGrandChildAggr = 0;
                DTreeNode* dTreeChild = dTreeNode->_firstChild;
                dTreeNode->_numOfDescendentAggr = 0;
                for (int child = 0; child < numberOfChildren; ++child)
                {
                        /* recursive call */
                        computeAggregateVectorInfo(dTreeChild);
                        /* get index for each aggregate of dTreeNode, by iterating
                         * over the "traversal" of each dTreeChild to get a
                         * consistent ordering of nodes
                         */
                        for (int i = 0; i < dTreeChild->_traversal.size(); ++i)
                        {
                                /* get child id and corresponding child node */
                                int grandChildID = dTreeChild->_traversal[i];
                                DTreeNode* grandChild =
                                        _dTree->getNode(grandChildID);
                                /* get the size of vector for this child's
                                 * aggregates */
                                if (_categoricalFeature[grandChildID])
                                {
                                        sizeOfGrandChildAggr =
                        grandChild->_numCatValues;
                                }
                                else // conts or non feature
                                {
                                        sizeOfGrandChildAggr = 1;
                                }
                                /* add the amended index to our map/traversal */
                                dTreeNode->_traversal.push_back(grandChildID);
                                dTreeNode->_
                        indexOfAttrInNodeAggregates.insert({grandChildID, index});
                                /* update our index for next grandChild, using the
                                 * values from the final grandchild */
                                index += sizeOfGrandChildAggr;
                                dTreeNode->_numOfDescendentAggr +=
                                                        sizeOfGrandChildAggr;
                        }
                        dTreeChild = dTreeChild->_next;
                }
                /* finally, we add the index for the parent under consideration
                 * which will be the final index after all children/
                 * grandchildren have been processed */
                dTreeNode->_traversal.push_back(nodeID);
                dTreeNode->_indexOfAttrInNodeAggregates.insert({nodeID, index});
                if(_categoricalFeature[nodeID])
                        dTreeNode->_numOfDescendentAggr += dTreeNode->_numCatValues;
                else // conts or non feature
                        dTreeNode->_numOfDescendentAggr += 1;
        }
};

/*
 * CategoricalRegressionOverJoin::computeMultiplicity()
 *
 * reused from RegressionOverJoin::computeMultiplicity()
 *
 * determines cnt/cnt_j for each union struct of the factorisation
 */
void CategoricalRegressionOverJoin::computeMultiplicity
        (node::Union* factorisationNode, DTreeNode* dTreeNode, unsigned int multi)
{
        unsigned int numberOfValues = factorisationNode->numberOfValues;
        int numberOfChildren = dTreeNode->_numOfChildren;
        node::Union** children = factorisationNode -> children;
        factorisationNode -> multiplicity += multi; // o/a caching...
        int nodeID = dTreeNode->_id;
        double* values = factorisationNode->values;
        for (unsigned int val = 0; val < numberOfValues; ++val)
        {
                DTreeNode* dTreeChild = dTreeNode->_firstChild;

                unsigned int localCount = 1;
                for (int child = 0; child < numberOfChildren; ++child)
                {
                        localCount *= children[val*numberOfChildren+child]->count;
                }

                for (int child = 0; child < numberOfChildren; ++child)
                {
                        unsigned int multiplicity = multi * localCount /
                                children[val*numberOfChildren+child]->count;
                        computeMultiplicity(children[val * numberOfChildren + child],
                                                     dTreeChild, multiplicity);
                        dTreeChild = dTreeChild->_next;
                }
        }
}

/*
 * CategoricalRegressionOverJoin::computeS1()
 *
 * New method
 *
 * Computes the aggregate vector S1 for each union struct, taking into account
 * whether each query variable is continuous, categorical or excluded as a model
 * feature.
 */
void CategoricalRegressionOverJoin::computeS1
                        (node::Union* factorisationNode, DTreeNode* dTreeNode)
{
        if (dTreeNode->_caching)
        {
                if (_S1Cached[factorisationNode->cacheIndex])
                        return;
        }

        int nodeID = dTreeNode->_id;
        unsigned int numberOfValues = factorisationNode->numberOfValues;
        double *values = factorisationNode->values;
        int numberOfChildren = dTreeNode->_numOfChildren;
        node::Union** children = factorisationNode->children;

        /* size of aggregate vectors for any given node will be
         * numOfDescendentAggr, for both conts and cat */
        factorisationNode->S1 = new double[dTreeNode->_numOfDescendentAggr];
        /* and initialise it to zero values */
        std::fill_n(factorisationNode->S1, dTreeNode->_numOfDescendentAggr, 0.0);
        /* create the array for Grad too, since same size */
        factorisationNode->Grad = new double[dTreeNode->_numOfDescendentAggr];
        std::fill_n(factorisationNode->Grad, dTreeNode->_numOfDescendentAggr, 0.0);

        /* we will also need to know the index of dTreeNode's own attribute
         * within the traversal of dTreeNode (i.e. at the end of it!) */
        int attrInNodeIndex = dTreeNode->_indexOfAttrInNodeAggregates[nodeID];
        if (numberOfChildren == 0) /* leaf node of factorisation */
        {
                if (_categoricalFeature[nodeID])
        {
                for (unsigned int val = 0; val < numberOfValues; ++val)
                {
                        int catAggrIndex = _catAggrToOverallAggrMapping
                                [_nodeIDToCatAggrMapping[nodeID]][values[val]];
                        factorisationNode->S1[0 + catAggrIndex] = 1.0;
                }
        }
        else if (_continuousFeature[nodeID])
        {
        for (unsigned int val = 0; val < numberOfValues; ++val)
                {
                        /* remember each value occurs once in given leaf,
                         * and there is only one attribute per leaf */
                        factorisationNode->S1[0] += values[val];
                }
        }
        else if (_nonFeature[nodeID])
        {
        for (unsigned int val = 0; val < numberOfValues; ++val)
                {
                        /* do nothing - want 0; */
                }
        }
}
else    /* our node has children, so need to combine the sums of the
          * "descendent" attributes in the d tree and also create new sums
          * for our current attribute in the d-tree - same code applies
          * for conts or cat variables */
{
        for (unsigned int val = 0; val < numberOfValues; ++val)
        {
                DTreeNode* dTreeChild = dTreeNode->_firstChild;
                int index = 0;
                for (int child = 0; child < numberOfChildren; ++child)
                {
                        /* recursive call */
                        computeS1(children[val * numberOfChildren + child],
                                                        dTreeChild);
                        children[val * numberOfChildren + child]->multiplier =
                                values[numberOfValues+val] /
                children[val * numberOfChildren + child]->count;
                        for (int S1Index = 0;
                                S1Index < dTreeChild->_numOfDescendentAggr;
                                        ++S1Index)
                        {
                                factorisationNode->S1[index] +=
                children[val * numberOfChildren + child]->S1[S1Index] *
                        children[val * numberOfChildren + child]->multiplier;
                                ++ index;
                        }

                        dTreeChild = dTreeChild->_next;
                }
                /* now the processing for the current attribute */
                if (_categoricalFeature[nodeID])
                {
                        /* index of this attribute is no longer 0 */
                        int catAggrIndex = _catAggrToOverallAggrMapping[
                                _nodeIDToCatAggrMapping[nodeID]][values[val]];
                        factorisationNode->S1[attrInNodeIndex + catAggrIndex]
```

```
                                        += 1.0 * values[numberOfValues + val];
                        }
                        else if (_continuousFeature[nodeID])
                        {
                                //
                                factorisationNode->S1[attrInNodeIndex]
                                        += values[val] * values[numberOfValues + val];
                        }
                        else if (_nonFeature[nodeID])
                        {
                                /* do nothing */
                        }
                }
        } /* end of else */
        /* mark that the values for a cached node are now in place */
        if (dTreeNode->_caching)
                _S1Cached[factorisationNode->cacheIndex] = true;
};

/*
 * CategoricalRegressionOverJoin::computeS1w()
 *
 * New method
 *
 * Computes the scalar S1w for each union struct, taking into account
 * whether each query variable is continuous, categorical or excluded as a model
 * feature.
 */
void CategoricalRegressionOverJoin::computeS1w(
                node::Union* factorisationNode, DTreeNode* dTreeNode)
{
        if (dTreeNode->_caching)
        {
                if (_S1wCached[factorisationNode->cacheIndex])
                        return;
        }

        int nodeID = dTreeNode->_id;
        unsigned int numberOfValues = factorisationNode->numberOfValues;
        double *values = factorisationNode->values;
        int numberOfChildren = dTreeNode->_numOfChildren;
        node::Union** children = factorisationNode->children;

        factorisationNode->S1w = 0;
        /* we will need to know the parameter corresponding to this nodeID
         * using the index of where the node would appear in the root node
         * of the dTree */
        int nodeInParamsIndex = dTreeNode->_indexOfNodeInRootAggregates;
        /* we will also need to know the index of dTreeNode's on attribute
         * within the traversal of dTreeNode (i.e. at the end of it!) */
        int attrInNodeIndex = dTreeNode->_indexOfAttrInNodeAggregates[nodeID];
        if (numberOfChildren == 0)
        {
                if (_categoricalFeature[nodeID])
                {
                        for (unsigned int val = 0; val < numberOfValues; ++val)
                        {
                                /* the place where each value is recorded (as a
                                 * "count" value of 1) is the index of the cat
                                 * value within that attribute (we know the index
                                 * of the attribute itself will be 0, as leaf is
```

```
                         * only a single attribute */
                        int catAggrIndex = _catAggrToOverallAggrMapping[
                                _nodeIDToCatAggrMapping[nodeID]][values[val]];
                        factorisationNode->S1w += 1.0 *
                                _parameters[nodeInParamsIndex + catAggrIndex];
                }
        }
        else if (_continuousFeature[nodeID])
        {
                for (unsigned int val = 0; val < numberOfValues; ++val)
                {
                        // the index of the attribute itself is again 0
                        factorisationNode->S1w += values[val] *
                                _parameters[nodeInParamsIndex];
                }
        }
        else if (_nonFeature[nodeID])
        {
                /* want to do nothing */
        }
}
else    /* our node has children, so need to combine the sums of the
         * "descendent" attributes in the d tree and also create new
         * sums for our current attribute in the d-tree - doesn't
         * matter if conts or cat variable */
{
        for (unsigned int val = 0; val < numberOfValues; ++val)
        {
                DTreeNode* dTreeChild = dTreeNode->_firstChild;
                int index = 0;
                for (int child = 0; child < numberOfChildren; ++child)
                {
                        /* recursive call */
                        computeS1w(children[val * numberOfChildren + child],
                                        dTreeChild);
                        factorisationNode->S1w +=
                                children[val * numberOfChildren + child]->S1w *
                children[val * numberOfChildren + child]->multiplier;
                        dTreeChild = dTreeChild->_next;
                }
                /* then do the "current" attribute we will need where
                 * this attribute appears in the list of parameters -
                 * this is simply the inded of the node, plus the index
                 * of where the node attribute appears within that node
                 * - both of which we know already */
                if (_categoricalFeature[nodeID])
                {
                        /* index of this attribute is no longer 0 -
                         * not a leaf */
                        int catAggrIndex =
                                _catAggrToOverallAggrMapping[
                _nodeIDToCatAggrMapping[nodeID]][values[val]];
                        factorisationNode->S1w += 1.0 *
                                values[numberOfValues + val] *
        _parameters[nodeInParamsIndex + attrInNodeIndex + catAggrIndex];
                }
                else if (_continuousFeature[nodeID])
                {
                        factorisationNode->S1w += values[val] *
                                values[numberOfValues + val] *
        _parameters[nodeInParamsIndex+attrInNodeIndex];
```

```
                        }
                        else if (_nonFeature[nodeID])
                        {
                                /* do nothing */
                        }
                }
        } /* end of else */
        /* mark that the values for a cached node are now in place */
        if (dTreeNode->_caching)
                _S1wCached[factorisationNode->cacheIndex] = true;
}

/*
 * CategoricalRegressionOverJoin::computeGrad()
 *
 * New method
 *
 * Computes the aggregate vector Grad for each union struct, taking into account
 * whether each query variable is continuous, categorical or excluded as a model
 * feature.
 */
void CategoricalRegressionOverJoin::computeGrad
                (node::Union* factorisationNode, DTreeNode* dTreeNode)
{
        int nodeID = dTreeNode -> _id;

        if (dTreeNode->_caching)
        {
                if (_GradCached[factorisationNode->cacheIndex])
                        return;
        }

        /* we already have S1, S1w, mulitiplier, multiplity and
         *count (= numberOfValues) for the factorisation node */

        unsigned int numberOfValues = factorisationNode->numberOfValues;
        node::Union** children = factorisationNode->children;
        int numberOfChildren = dTreeNode -> _numOfChildren;
        unsigned int multiplicity = factorisationNode->multiplicity;
        double* values = factorisationNode->values;

        /* reset our Grad vectors to 0.0 */
        memset(factorisationNode->Grad, 0, sizeof(double)*dTreeNode->
                                _numOfDescendentAggr);

        /* for Grad we will need to know the parameter corresponding
         * to this nodeID using the index of where the node would appear
         * in the root node of the dTree */
        int nodeInParamsIndex = dTreeNode->_indexOfNodeInRootAggregates;
        /* we will also need to know the index of dTreeNode's own attribute
         * within the traversal of dTreeNode (i.e. at the end of it -
         * if it's categorical then we'll also be adding the value
         * index too!!) */
        int attrInNodeIndex = dTreeNode->_indexOfAttrInNodeAggregates[nodeID];

        if (numberOfChildren == 0) // leaf node of factorisation tree
        {
                if (_categoricalFeature[nodeID])
                {
                        for (unsigned int val = 0; val < numberOfValues; ++val)
                        {
```

```
                int catAggrIndex = _catAggrToOverallAggrMapping[
    _nodeIDToCatAggrMapping[nodeID]][values[val]];
                factorisationNode->Grad[0 + catAggrIndex] += 1.0 *
    (_parameters[nodeInParamsIndex + catAggrIndex] * 1.0);
            }
    }
    else if (_continuousFeature[nodeID])
    {
    for (unsigned int val = 0; val < numberOfValues; ++val)
            {
                factorisationNode->Grad[0] += values[val] *
                    _parameters[nodeInParamsIndex] * values[val];
            }
    }
    else if (_nonFeature[nodeID])
    {
    for (unsigned int val = 0; val < numberOfValues; ++val)
            {
                // do nothing - leave Grad = 0;
            }
    }
}
else /* Not a leaf node. */
{
    /* we need an array to hold our local S1w values so we can
     * calculate S in our next pass over the values/children */
    double localS1w[numberOfValues];
    std::fill_n(localS1w, numberOfValues, 0.0);
    for (unsigned int val = 0; val < numberOfValues; ++val)
    {
        DTreeNode* dTreeChild = dTreeNode->_firstChild;
        /* we have to iterate through the children of this node
         * twice - the first time to calculate S1w for this value
         * values[val] */
        for (int child = 0; child < numberOfChildren; ++child)
        {
            localS1w[val] +=
        children[val * numberOfChildren + child]->S1w *
    children[val * numberOfChildren + child]->multiplier;
            dTreeChild->_next;
        }
    }
    for (unsigned int val = 0; val < numberOfValues; ++val)
    {
        /* we will just need to be careful as to whether A is
         * categorical or a non-feature, as this will determine
         * the index of the parameters required for wA and the
         * position of A in S1 for the factorisationNode */
        int paramIndexToUse;
        int aIndexToUse;
        int catAggrIndex;

        /* we can now iterate over our children again, using
         * localS1w to calculate S */
        DTreeNode* dTreeChild = dTreeNode->_firstChild;
        int index = 0;
        for (int child = 0; child < numberOfChildren; ++child)
        {
            /* call our function recursively on each of our child
             * factorisation nodes */
            computeGrad(
children[val * numberOfChildren + child], dTreeChild);

       /* calculate S for each child node */
       double localS = localS1w[val] -
              children[val * numberOfChildren + child]->S1w *
children[val * numberOfChildren + child]->multiplier;
       /* now iterate over Grad/S1 */
       for (int gradIndex = 0;
              gradIndex < dTreeChild->_numOfDescendentAggr;
                     ++gradIndex)
       {
              if(_categoricalFeature[nodeID])
              {
                     catAggrIndex =
                            _catAggrToOverallAggrMapping[
              _nodeIDToCatAggrMapping[nodeID]][values[val]];
                     paramIndexToUse = nodeInParamsIndex +
                            attrInNodeIndex + catAggrIndex;
                     aIndexToUse = attrInNodeIndex +
              catAggrIndex;
                     /* we can now do Grad for everything
                      * except the 'current' attribute,
                      * using info we have already: */
                     factorisationNode->Grad[index] +=
                            /* Grad * multiplier */
children[val * numberOfChildren + child]-> Grad[gradIndex] *
children[val * numberOfChildren + child]->multiplier +
                            /* S1(j)[B]/cnt(j) */
children[val * numberOfChildren + child]->
S1[gradIndex] * localS /
children[val * numberOfChildren + child]->count +
                            /* wA * a * cnt(a) * S1(j)[a] */
_parameters[paramIndexToUse] * 1.0 *
values[val + numberOfValues] *
children[val * numberOfChildren + child]->S1[gradIndex] /
children[val * numberOfChildren + child]->count;
              }
              else if(_continuousFeature[nodeID])
              {
                     paramIndexToUse = nodeInParamsIndex +
                                   attrInNodeIndex;
                     aIndexToUse = attrInNodeIndex;
                     /* again do Grad for everything
                      * except the 'current' attribute,
                      * using info we have already: */
                     factorisationNode->Grad[index] +=
                            /* Grad * multiplier */
children[val * numberOfChildren + child]->
Grad[gradIndex] *
children[val * numberOfChildren + child]->multiplier +
                            /* S1(j)[B]/cnt(j) */
children[val * numberOfChildren + child]->
S1[gradIndex] * localS /
children[val * numberOfChildren + child]->count +
                            /* wA * a * cnt(a) * S1(j)[a] */
_parameters[paramIndexToUse] *
values[val] * values[val + numberOfValues] *
children[val * numberOfChildren + child]->S1[gradIndex] /
children[val * numberOfChildren + child]->count;
              }
              else if(_nonFeature[nodeID])
```

```
                                        {
                                                paramIndexToUse = nodeInParamsIndex +
                                                                        attrInNodeIndex;
                                                aIndexToUse = attrInNodeIndex;
                                                /* again do Grad for everything
                                                 * except the ’current’
                                                 * attribute, using info
                                                 * we have already: */
                                                factorisationNode->Grad[index] +=
                                                        /* Grad * multiplier */
                        children[val * numberOfChildren + child]->
                        Grad[gradIndex] *
                        children[val * numberOfChildren + child]->
                                                                multiplier +
                                                        /* S1(j)[B]/cnt(j) */
                        children[val * numberOfChildren + child]->
                        S1[gradIndex] * localS /
                        children[val * numberOfChildren + child]->count;
                                        }
                                        ++index;
                                }
                                dTreeChild = dTreeChild->_next;
                        }
                        /* then do the "current" attribute: */
                        if (_categoricalFeature[nodeID])
                        {
                                int catAggrIndex = _catAggrToOverallAggrMapping[
                                        _nodeIDToCatAggrMapping[nodeID]][values[val]];
                                int paramIndexToUse = nodeInParamsIndex +
                                        attrInNodeIndex + catAggrIndex;
                                int aIndexToUse = attrInNodeIndex + catAggrIndex;
                                factorisationNode->Grad[aIndexToUse] += 1.0 *
                                        (_parameters[paramIndexToUse] * 1.0 *
                                                values[numberOfValues + val] +
                                localS1w[val]);
                        }
                        else if (_continuousFeature[nodeID])
                        {
                                factorisationNode->Grad[aIndexToUse] += values[val] *
                                        (_parameters[paramIndexToUse] * values[val] *
                                                values[numberOfValues + val] +
                                localS1w[val]);
                        }
                        else if (_nonFeature[nodeID])
                        {
                                /* do nothing */
                        }
                }
        }
        /* If this node allows for caching, we fill the cached aggreates
         *with the computed aggregates for this node. */
        if (dTreeNode->_caching)
        {
                _GradCached[factorisationNode->cacheIndex] = true;
        }
};

/*
 * CategoricalRegressionOverJoin::computeGradient()
 *
 * Modified from RegressionOverJoin::computeGradient to accommodate the
 * new methods for computing aggregate vectors.
 *
 * carries out one convergence step of the batch gradient descent process.
 */
void CategoricalRegressionOverJoin::computeGradient()
{
       int dTreeRootId = _dTree->_root->_id;

       memset(_S1wCached, 0,
              sizeof(bool)*_joinEngine->getNumberOfCachedValues());
       memset(_GradCached, 0,
              sizeof(bool)*_joinEngine->getNumberOfCachedValues());
       memset(_gradient, 0,
              sizeof(double)*NUM_OF_PARAMETERS);

       /* need to recompute S1w and Grad with latest parameters */
       computeS1w(rootNode, _dTree->_root);
       computeGrad(rootNode, _dTree->_root);
       /* gradient[j] = (Grad[j] + S1[j]*w_0)/numOfTuples
        * so copy Grad to gradients then perform addition and division */
       memcpy(_gradient, rootNode->Grad, sizeof(double)*NUM_OF_PARAMETERS);
       for (int aggr = 0; aggr < _dTree->_root->_numOfDescendentAggr; ++aggr)
       {
              _gradient[aggr] += rootNode->S1[aggr] *
                             _parameters[NUM_OF_PARAMETERS - 1];
              _gradient[aggr] /= rootNode->count;
       }

       /* _gradient[0] = (S1w +w_0*numOfTuples)/numOfTuples */
       _gradient[NUM_OF_PARAMETERS - 1] = (rootNode->S1w +
              rootNode->count * _parameters[NUM_OF_PARAMETERS-1]) /
                     rootNode->count;

       BINFO("\n CURRENT STUFF---------------------------------\n")
       BINFO("current S1[");
       for (size_t param = 0; param < NUM_OF_PARAMETERS; ++param)
       {
              BINFO(" "+to_string(rootNode->S1[param]));
       }
       BINFO("\n current parameter[");
       for (size_t param = 0; param < NUM_OF_PARAMETERS; ++param)
       {
              BINFO(" "+to_string(_parameters[param]));
       }
       BINFO("\n current S1w "+to_string(rootNode->S1w));
       BINFO("\n current Grad[");
       for (size_t param = 0; param < NUM_OF_PARAMETERS; ++param)
       {
              BINFO(" "+to_string(rootNode->Grad[param]));
       }
       BINFO("\n current gradient[");
       for (size_t param = 0; param < NUM_OF_PARAMETERS; ++param)
       {
              BINFO(" "+to_string(_gradient[param]));
       }
       BINFO("]\n");
       };

/*
 * CategoricalRegressionOverJoin::computeS1()
 *
 * Based closely on RegressionOverJoin::optimize() but using simple adaptive
 * step size for batch gradient descent.
 *
 * Completes the convergence steps for the batch gradient descent process
 * to learn the model parameters
 */
void CategoricalRegressionOverJoin::optimize()
{
        /* firstGradientNorm is used in the normalised stopping condition:
         * gradientNorm / (firstGradientNorm + epsilon) */
        double firstGradientNorm = 0.0;

        /* Need an initial set of gradients based on initial parameters
         * that will be used for convergence testing */

#ifdef BENCH
        int64_t startOneGradient = duration_cast<milliseconds>(
                        system_clock::now().time_since_epoch()).count();
#endif

        computeGradient();

#ifdef BENCH
        int64_t timeOneGradient = duration_cast<milliseconds>
                (system_clock::now().time_since_epoch()).count() -
                        startOneGradient;

        /* Write First Gradient Computation times to times file */
    std::ofstream ofs("times.txt", std::ofstream::out | std::ofstream::app);

        if (ofs.is_open())
                ofs << "convergence: optimize - one gradient: \t"+
                        to_string(timeOneGradient) + "\n";
        else
                cout << "Unable to open times.txt \n";
        ofs.close();
#endif

DINFO("NUM_OF_PARAMETERS: "+to_string(NUM_OF_PARAMETERS) + "\n");
BINFO(
        "MASTER - convergence: optimize - one gradient computation: " +
                to_string(timeOneGradient) + "ms.\n");

for (int i = 0; i < NUM_OF_PARAMETERS - 1; ++i)
                DINFO("Grad aggregate["+to_string(i)+"] : "+
                        to_string(rootNode->Grad[i])+" \n");

DINFO("\n");
for (size_t i = 0; i < NUM_OF_PARAMETERS; ++i)
        DINFO("gradient["+to_string(i)+"] : "+to_string(_gradient[i])+" \n");

DINFO("\n");

        /* Simple gradient descent step */
        for (size_t j = 0; j < NUM_OF_PARAMETERS; ++j)
        {
        /* do not change label parameter: constrained to be -1 */
                if (j == LABEL_PARAMETER_INDEX) continue;

                /* _previousParameters and _previousGradients are used to
                * compute the stepSize */
        _previousParameters[j] = _parameters[j];
        _previousGradients[j] = _gradient[j];

        /* firstNorm is the (norm of Gradient)^2 */
        firstGradientNorm += _gradient[j] * _gradient[j];

        /* Update Parameters in direction of gradient */
        _parameters[j] = _parameters[j] - stepSize * _gradient[j];
}

// alpha = nextAlpha;
++_numOfIterations;
firstGradientNorm = sqrt(firstGradientNorm);

while (!_converged && _numOfIterations < 200)
{
        /* Compute the gradient based on latest parameter values */
        computeGradient();

        DINFO("grad: ");
        for (size_t j = 0; j < NUM_OF_PARAMETERS; ++j)
                DINFO(to_string(_gradient[j]) + " ")
        DINFO("\n");

        DINFO("params: ");
        for (size_t j = 0; j < NUM_OF_PARAMETERS; ++j)
                DINFO(std::to_string(_parameters[j]) + " ")
        DINFO("\n");

        /* Update the stepSize */
        stepSize = computeStepSize();

        /* The norm of the gradients is used both in the stopping
         * condition and backtracking (not used here). */
        double gradientNorm = 0.0;

        /* Update the parameters. */
        for (size_t j = 0; j < NUM_OF_PARAMETERS; ++j)
        {
                /* again, don’t update label parameters */
                if (j == LABEL_PARAMETER_INDEX) continue;

                gradientNorm += _gradient[j] * _gradient[j];

                /* these are used to compute the adaptive spectral
                 * stepsize at iteration k+1 */
    _previousParameters[j] = _parameters[j];
                _previousGradients[j] = _gradient[j];

                /* Updates parameter j based in in direction of
                 * gradient (regularization not used here. */
                _parameters[j] = _parameters[j] - stepSize * _gradient[j];
        }

        DINFO("grad: ");
        for (size_t j = 0; j < NUM_OF_PARAMETERS; ++j)
                DINFO(to_string(_gradient[j]) + " ")
        DINFO("\n");

        DINFO("param: ");
        for (size_t j = 0; j < NUM_OF_PARAMETERS; ++j)
```

```
                        DINFO(to_string(_parameters[j]) + " ")
                DINFO("\n");

                if (sqrt(gradientNorm) / (firstGradientNorm + 0.01) < _epsilon)
                {
                        _converged = true;
                        break;
                }

                ++_numOfIterations;
                cout << "number of iterations " << _numOfIterations << endl;
        }
};
```

Appendix B

Methods added to CategoricalFactorisedJoin.cpp

This Appendix contains the new methods added to the CategoricalFactorisedJoin class. The structure of the class was modelled on an existing class for another regression method (FactorisedJoin), but extended with new/modified methods. Comments before each method indicate the extent to which existing code has been reused. Methods that are unchanged in the CategoricalFactorisedJoin class are not listed here.

```
/*
 * CategoricalFactorisedJoin::sortDataForCategoricalVariables()
 *
 * Modified from FactorisedJoin::sortDataToProcess to allow sorting on
 * a single categorical variable.
 *
 * sorts each of the database tables on the column corresponding to the
 * variable of the d-tree node passed as argument.
 */

void CategoricalFactorisedJoin::sortDataForCategoricalValues(DTreeNode* node)
{

        /* Contains an ID for each node in the DTree and for each table. */
        vector<vector<int>> treeAttributeIDs(NUM_OF_TABLES);

        /* Iterate through all tables. */
        for (size_t table = 0; table < NUM_OF_TABLES; ++table)
        {
                /* Iterate through all the table attribute IDs of the current table. */
                for (uint_fast16_t id : _dataHandler->getTableAttributes()[table])
                {
                        /* Scan through the (string, ID) mapping to find the attribute
                         * name of the current attribute ID. */
                        for (auto pair : _dataHandler->getNamesMapping())
                        {
                                if (pair.second == id)
                                {
                                        /* Retrieve DTree ID with attribute name. */
                                        auto index = _dTree->getIndexByName(pair.first);

                                        treeAttributeIDs[table].push_back(index);
                                        break;
                                }
                        }
                }
        }
        /*
         * The ids vectors will contain the pairs of INDEXES in the array
         * attributes-attr. For each attribute KEY-index we append the pairs of
         * indexes having that key, so we will have all the pairs of
         * <relationIndex-attributeIndex> for each attribute KEY.
         */
        _ids = new vector<pair<int, int> > [NUM_OF_ATTRIBUTES];
        for (size_t i = 0; i < NUM_OF_ATTRIBUTES; ++i)
        {
                for (size_t j = 0; j < NUM_OF_TABLES; ++j)
                {
                        for (size_t k = 0; k < treeAttributeIDs[j].size(); ++k)
                                if (treeAttributeIDs[j][k] == (int) i)
                                {
                                        _ids[i].push_back(make_pair(j, k));
                                        break;
                                }
                }
        }

        /* we only need to think about our single node, which we know is an
         * attribute, and we know is categorical (else this method would
         * not have been invoked in the first place) */
        vector<int_fast16_t> priority;
        for (size_t table = 0; table < NUM_OF_TABLES; ++table)
        {
                priority.clear();

                int_fast16_t pos = -1;
                for (size_t j = 0; j < treeAttributeIDs[table].size(); ++j)
                        if (node->_id == treeAttributeIDs[table][j])
                        {
                                pos = j;
                                break;
                        }
                if (pos != -1)
                {
                        priority.push_back(pos);

                        /* Launch sorting; depending on the compiler, this will or will
                         * not be done in parallel. nb put into if loop, as we're not
                         * going to sort if the attribute not in the table!! */
                        sortingAlgorithm(_data[table].begin(), _data[table].end(),
                                        ValueOrdering(priority.data(), priority.size()));
                }
        }
}

/*
 * CategoricalFactorisedJoin::getCategoricalValues()
 *
 * new method
 *
 * joins database tables on just the relation columns for a single categorical
 * variable, to get the values that this attribute can take, and which we can
 * then use in the aggregates. */
double* CategoricalFactorisedJoin::getCategoricalValues(DTreeNode* node) {

        double* result;
        // method code based on run() method...
        if (_numOfThreads > 1)
        {
                cout << "The multi-threaded version is currently not supported. \n";
                cout << "Continuing with 1 thread. \n";
                _numOfThreads = 1;
        }

        /* Spawn threads and do partitioning. */
        if (_numOfThreads == 1)
        {
        _partitionFields2 = new PartitionAttributesCatValues[1];
                /* Set bounds for the different tables: they cover
                 * the whole tables */
                int* lowerBounds = new int[NUM_OF_TABLES];
                int* upperBounds = new int[NUM_OF_TABLES];
                for (size_t table = 0; table < NUM_OF_TABLES; ++table)
                {
                        lowerBounds[table] = 0;
                        upperBounds[table] = _data[table].size() - 1;
                }

                /* Launch our amended runAggregator to find categorical
                 * values for this node */
                result =
                        runCategoricalValuesFinder(node, lowerBounds, upperBounds, 0);

                /* Clean up bound arrays. */
                delete[] lowerBounds;
                delete[] upperBounds;
                /* and invoke *destructor* on our PartitionAggregateCatValues
                 * data structure */
                delete[] _partitionFields2;
        }
        return result;
}

/*
 * CategoricalFactorisedJoin::runCategoricalValuesFinder()
 *
 * Modified from FactorisedJoin::runAggregator to perform leapfrogging
 * join only on each categorical variable. Uses a new data structure
 * defined in CategoricalFactorisedJoin.h
 *
 */
double* CategoricalFactorisedJoin::runCategoricalValuesFinder
                (DTreeNode* node, int* lowerBounds, int* upperBounds,
                        unsigned int partition) {
        /* we're not going to use all the partition fields, just the ones
         * we need so define a new data structure :-) */

        /* Initialize varMap to required size and set entries to zero. */
        _partitionFields2[partition].LOWERBOUND =
                                            new int*[NUM_OF_ATTRIBUTES];
        _partitionFields2[partition].UPPERBOUND =
                                            new int*[NUM_OF_ATTRIBUTES];
        _partitionFields2[partition].catValues =
                                            new std::vector<double>[1];

        for (size_t i = 0; i < NUM_OF_ATTRIBUTES; ++i)
        {
                _partitionFields2[partition].LOWERBOUND[i] =
                                            new int[NUM_OF_TABLES];
                _partitionFields2[partition].UPPERBOUND[i] =
                                            new int[NUM_OF_TABLES];
        }

        _partitionFields2[partition].ordering =
                                            new int*[NUM_OF_ATTRIBUTES];
        for (size_t i = 0; i < NUM_OF_ATTRIBUTES; ++i)
                _partitionFields2[partition].ordering[i] =
                                            new int[_ids[i].size()];

        DINFO("Before leapfrogging join for categorical values! \n");
        /* This performs the join and recursively updates the
         * regression aggregates. */
        return leapfroggingJoinSingleAttribute
                      (node, lowerBounds, upperBounds, partition);
}

/*
 * CategoricalFactorisedJoin::leapfroggingJoinSingleAttribute()
 *
 * Modified from FactorisedJoin::leapfroggingJoin() to allow joining on
 * a single categorical variables.*
 *
 * performs leapfrogging join for all database tables on each categorical
 * variable.
 */
CategoricalFactorisedJoin::leapfroggingJoinSingleAttribute
               (DTreeNode* node, int* lower, int* upper, unsigned int partition)
{
        int_fast16_t nodeID = node->_id;

        /* lower range pointer for each relation. */
        int* l = _partitionFields2[partition].LOWERBOUND[nodeID];
        memcpy(l, lower, sizeof(int) * NUM_OF_TABLES);

        /* upper range pointer for each relation. */
        int* u = _partitionFields2[partition].UPPERBOUND[nodeID];
        memcpy(u, upper, sizeof(int) * NUM_OF_TABLES);

        vector<double>& unionValues =
                      _partitionFields2[partition].catValues[0];
        unionValues.clear();
        /* reserve spaces for the nodeID, and eventual
           number of categorical values */
        unionValues.push_back(nodeID);
        unionValues.push_back(0);

        int* ordering = _partitionFields2[partition].ordering[nodeID];

        /* Provides order of Relations from min -> max. */
        getRelationOrdering(l, node, ordering);

        /* Value that satisfies the join query - set in seekValue. */
        double val;
        unsigned int count = 0;

        /* Indexes used by the join algorithm. */
        int i, j, rel = 0, numOfRel = _ids[nodeID].size();
        bool atEnd = false;
        while (!atEnd)
```

```
{
        /* seek the value that satisfies the join query */
        atEnd =
                seekValue(node, rel, ordering, numOfRel, l, upper, val);

        if (atEnd)
                break;

        /* Get range of tuples with value equal to val. */
        for (size_t k = 0; k < _ids[nodeID].size(); ++k)
        {
                i = _ids[node->_id][k].first;
                j = _ids[node->_id][k].second;
                u[i] = l[i];

                while (u[i] < upper[i] && _data[i][u[i] + 1][j] == val)
                {
                        ++u[i];
                }
        }

        /*
         * In leapfroggingJoin aggregates are updated based on the
         * value that satisfied the join query. Here, assuming that
         * we don't have any dangling tuples, we just want to log
         * the values of this node that we have seen!
         */

        /* Push back the value from this node - remember we already
         * have a space reserved for nodeID and another for the count */
        unionValues.push_back(val);
        ++count;

        for (size_t k = 0; k < _ids[nodeID].size(); ++k)
        {
                i = _ids[nodeID][k].first;
                l[i] = u[i];
        }

        i = _ids[nodeID][ordering[rel]].first;
        l[i] += 1;

        if (l[i] > upper[i])
        {
                atEnd = true;
                break;
        }
        else
        {
                rel = (rel + 1) % numOfRel;
        }
}
assert (count == unionValues.size());

/* Copy values into array - don't forget the extra 0 at the
 * front to hold the count */
double* result = new double[count+2];
memcpy(result, &unionValues[0],(count + 2)*sizeof(double));
result[1] = count;

return result;
```

```
}
```

References

[1] S. (Serge) Abiteboul, Richard Hull, and Victor. Vianu. *Foundations of databases.* Addison-Wesley, 1995.

[2] A.E.S. Why firms are piling into artificial intelligence. *The Economist Explains[Blog]*, apr 2016.

[3] Albert Atserias, Martin Grohe, and Dániel Marx. Size Bounds and Query Plans for Relational Joins. *SIAM J. Comput.*, 42(4):1737–1767, jan 2013.

[4] Nurzhan Bakibayev, Tomáš Kočiský, Dan Olteanu, and Jakub Závodný. Aggregation and Ordering in Factorised Databases. *Proc. VLDB Endow.*, 6(14):1990–2001, jul 2013.

[5] Nurzhan Bakibayev, Dan Olteanu, and Jakub Závodný. FDB: A query engine for factorised relational databases. *Vldb*, pages 1232–1243, mar 2012.

[6] Christopher M. Bishop. *Pattern recognition and machine learning.* Springer, 2006.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*, volume 212 Suppl. MIT Press, 2012.

[8] Tom Goldstein, Christoph Studer, and Richard Baraniuk. A Field Guide to Forward-Backward Splitting with a FASTA Implementation. *arXiv:1411.3406*, page 25, nov 2014.

[9] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Robbers, marshals, and guards: game theoretic and logical characterizations of hypertree width. *J. Comput. Syst. Sci.*, 66(4):775–808, jun 2003.

[10] Martin Grohe and Dániel Marx. Constraint solving via fractional edge covers. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms. SODA '06*, pages 289–298. Association for Computing Machinery, 2006.

[11] David Money. Harris and Sarah L. Harris. *Digital design and computer architecture.* Morgan Kaufmann Publishers, 2007.

[12] Joseph M Hellerstein, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Caleb Welton, Uc Berkeley Christoper, U Wisconsin Florian Schoppmann FlorianSchoppmann, Emccom Greenplum Daisy Zhe Wang, U Florida Eugene Fratkin, Greenplum Aleksander Gorajek AleksanderGorajek, Emccom Greenplum Kee Siong Ng, Emccom Greenplum Xixuan Feng, U Wisconsin Kun Li, and U Florida Arun Kumar. The MADlib Analytics Library or MAD Skills , the SQL. *Proc. VLDB Endow.*, pages 1700–1711, aug 2012.

[13] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. FAQ: Questions Asked Frequently. *Proc. 35th ACM SIGMOD-SIGACT-SIGAI Symp. Princ. Database Syst. - Pod. '16*, pages 13–28, apr 2016.

[14] Kevin P. Murphy. *Machine Learning: a Probabilistic Perspective.* MIT Press, 2012.

[15] Andrew Ng. *CS229 Lecture Notes.* Stanford and Coursera: http://cs229.stanford.edu/notes/cs229-notes1.pdf, 2014.

[16] Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. In-Database Learning with Sparse Tensors. *arXiv Prepr. arXiv1703.04780*, mar 2017.

[17] Dan Olteanu and Maximilian Schleich. F : Regression Models over Factorized Views. In *Proc.\ VLDB Endow.*, volume 9, pages 4–8. Association for Computing Machinery, 2016.

[18] Dan Olteanu and Maximilian Schleich. Factorized Databases. *ACM SIGMOD Rec.*, 45(2):5–16, sep 2016.

[19] Dan Olteanu and Jakub Závodný. Size Bounds for Factorised Representations of Query Results. *ACM Trans. Database Syst.*, 40(1):1–44, mar 2015.

[20] R Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2014.

[21] Christopher Ré, Divy Agrawal, Magdalena Balazinska, Michael Cafarella, Michael Jordan, Tim Kraska, and Raghu Ramakrishnan. Machine Learning and Databases: The Sound of Things to Come or a Cacophony of Hype? *Proc. 2015 ACM SIGMOD Int. Conf. Manag. Data*, pages 283–284, 2015.

[22] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. Learning Linear Regression Models over Factorized Joins. In *Proc. 2016 Int. Conf. Manag. Data - SIGMOD '16*, pages 3–18, New York, New York, USA, 2016. ACM Press.

[23] Todd L. Veldhuizen. Leapfrog Triejoin: a worst-case optimal join algorithm. oct 2012.

[24] Todd L. Veldhuizen. Triejoin: a simple, worst-case optimal join algorithm. In *Schweikardt, N., Christophides, V., Leroy, V. (Hrsg.) Proceedings of the 17th International Conference on Database Theory (ICDT)*, page 96–106. OpenProceedings.org, 2014.