University of Oxford

QR Decomposition of Normalised Relational Data

Bas A.M. van Geffen
Kellogg College

Supervisor
Prof. Dan Olteanu

A dissertation submitted for the degree of:
Master of Science in Computer Science

Trinity 2018

Abstract

The simultaneous rise of machine learning and big data has led to a pressing need to develop solutions that can process the large amounts of data available. In-database analytics is an important approach, because it avoids the expensive import-export step at the interface between the database system and the statistical package. Within in-database analytics, a technique known as factorised learning exploits the relational structure in the data to decompose the learning algorithm into a set of queries over the database relations. However, the current approach requires manually rewriting machine learning algorithms to operations that can be pushed past the join query.

This dissertation explores the in-database factorised setting at the more fundamental linear algebra level. In particular, we show that the QR decomposition of a matrix $\boldsymbol{A}$ defined by a join query over a relational database can be computed *without materialising* the join result $\boldsymbol{A}$.

We present the system **F-GS**, which implements a two-layered factorisation of a procedure used to compute QR decompositions. The first layer is an algebraic rewrite of the Gram-Schmidt process to terms expressible as aggregate queries. The second layer further pushes these aggregate queries past the join query defining $\boldsymbol{A}$.

F-GS provides both theoretical and practical improvements over the state of the art. We show that through utilising factorised computation and a sparse encoding for categorical data, **F-GS** can achieve asymptotically lower computational complexity than out-of-database alternatives. Moreover, experimental results confirm that the speedup of **F-GS** over popular alternatives matches the compression ratio brought by factorisation.

This dissertation introduces the first factorised matrix decomposition approach and shows that some of its applications, including solving linear least squares and obtaining a singular-value decomposition, can be performed without materialising the join result. By extension, our results open up a line of research into fundamental linear algebra operations in an in-database setting.

Acknowledgements

First of all, I would like to express my most sincere gratitude to my supervisor Prof. Dan Olteanu. His guidance and input over the last months were imperative to the outcome of this project. The feedback and push he provided turned this dissertation into something I am truly proud of.

I would also like to thank Maximilian Schleich for his guidance and suggestions for the research project starting from day one. His willingness to help and answer my questions regarding implementation and experiments is genuinely appreciated.

I am grateful to Fabian Peternek for his participation in our many discussions over the course of this project.

Finally, I would like to thank my parents and girlfriend for their support and encouragement throughout my studies.

Contents

Chapter 1

Introduction

Machine learning (ML) and data analytics are of growing importance in both academia and industry. Today, machine learning plays an important role in many industries, including retail, healthcare, finance, and manufacturing. Some prominent examples of applications are facial recognition systems (such as Apple's Face ID), voice recognition (e.g. Amazon's Alexa), and recommendation systems. For an example of the last, consider that Netflix claims that its recommendation system is responsible for 80% of streamed content. Moreover, Netflix values the combined effects of its personalisation and recommendation algorithms at 1 billion US dollars per year [1].

Simultaneously to the rise of machine learning, the term Big Data was coined to describe the enormous growth in data available and being generated. A commonly cited statistic (by IBM) claims that 90% of the world's data was generated in the two years prior [2]. Even though this increase in data has great value and impact (e.g. in retail), various challenges are associated with dealing with big data. In particular, visualising, analysing, or processing the data in some way becomes increasingly difficult. Similarly, performing machine learning on large datasets requires massive amounts of processing power and main memory capacity.

In practice, large datasets are stored in database management systems (DBMSs) as several relations. The data, which is kept *normalised* over many relations, can be viewed or reassembled as needed without needing to reorganise the relations. In order to perform machine learning, the join result is computed by the DBMS, which then exports the join result as a single large *design matrix* to the machine learning component. Moreover, modern software enterprise stacks (such as those provided by IBM, Oracle and Microsoft) provide specialised components for different tasks, including: data mining, online analytical processing, and business intelligence. Consequently, large scale ML pipelines are used to combine several systems, with significant time being spent at the interface between these different components. Additionally, this approach introduces significant costs in maintaining and operating the many different software solutions [3].

The expensive import and export steps at each interface contribute strongly to the scalability issues in machine learning. Moreover, the join result introduces large amounts of redundancy in the data, which affects the processing and export time at each component in the pipeline.

One approach to address the scalability issues of machine learning models is to split the dataset into partitions that are small enough to be processed. Instead of training one model on the entire dataset, each partition gets its own model. For example, a recommendation system can consist of one model per region, which is trained on and used for only one specific region. However, this approach can decrease the predictive power of a model, because some useful patterns may not be present in the partitioned dataset. Moreover, finding an appropriate partitioning both in terms of the partition sizes and loss of accuracy is non-trivial and requires domain expertise [4].

In-database analytics, which sets out to tightly integrate ML and analytics with the database system, offers an alternative approach to deal with the scalability issues. In-database analytics unifies the DBMS with the plethora of components into one single engine. A key realisation is that large parts of ML algorithms can be expressed as database queries [5]. This strategy retains a relational perspective of the data and offers many potential benefits in practical scenarios. First of all, it avoids the time-consuming import/export step at the interface between the specialised components in a conventional data pipeline. Secondly, mature DBMS technology for managing large datasets can be exploited and potentially extended. Hence, a unified engine can effectively address the current limitations of learning over large databases [3].

The conventional method to incorporate ML in the DBMS is to compute the design matrix as the listing representation of the DBMS query result. The ML algorithm is then performed inside the database, thus avoiding the time-consuming import/export step. A more promising way goes beyond this: it decomposes ML tasks into subtasks that are expressible as relational algebra queries, and subsequently further pushes these queries past the join that puts together the input relations. This approach, known as *factorised learning* in literature, is compelling because it can lower the computational complexity and lead to large performance benefits in practice [6][7][8].

This dissertation builds on the observation that many machine learning models at their core rely on a series of linear algebra operations, e.g. matrix decompositions, Gaussian elimination and the matrix inverse. Whereas earlier work considered the integration of a specific ML task (e.g. learning polynomial regression models) with the DBMS, our work lies at the more fundamental linear algebra level. The goal of this dissertation is to break new ground on in-database linear algebra. Ultimately, an in-database linear algebra library offers a systematic approach to reuse obtained results to develop or improve in-database algorithms.

We set out to show that fundamental linear algebra operations can be efficiently performed in the in-database factorised setting outlined, with the promised lower computational complexity and high performance improvements. Specifically, this dissertation describes a completely factorised approach to compute a fundamental matrix decomposition: the *QR decomposition*. The QR decomposition is an essential tool in any widely-used linear algebra library (including NumPy, R and MATLAB), because of its application in solving linear least squares and as the basis of an eigenvalue algorithm. More concretely, we consider the following problem setting:

Given a database $\mathbf{D}$ (consisting of continuous and categorical attributes) and a join query Q to define a design matrix $\boldsymbol{A}$, we would like to compute the factorised QR decomposition of $\boldsymbol{A}$ matrix *without materialising this matrix*, such that:

$$\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{R} = (\boldsymbol{A}\boldsymbol{C})\boldsymbol{R}$$

We propose an algebraic rewrite of the *Gram-Schmidt process*, a method used to compute QR decompositions, to aggregate queries over $Q(\mathbf{D})$. We show that the entries in $\boldsymbol{C}$ and $\boldsymbol{R}$ can be expressed in terms of inner products of the columns of $\boldsymbol{A}$. Similarly to prior work on factorised learning, we show that this computation can be performed without materialising $\boldsymbol{A}$. Our novel approach effectively pushes arithmetic computations necessary for the QR decomposition past the join of the input relations.

Incorporating heterogeneous data in the design matrix poses a second challenge, because the QR decomposition is defined for numerical matrices. Unlike continuous features, categorical variables are normally not aggregated together and admit no natural ordering. Therefore, the state-of-the-art approach is to use the one-hot encoding that is highly redundant. We obtain the same outcome while using a sparse encoding that was proposed within factorised learning [6].

1.1 Contributions

We introduce **F-GS**, a novel system for in-database factorised QR decompositions. The contributions of this dissertation are:

- This is the first system that shows how to approach a fundamental linear algebra operation, in particular the QR decomposition, using an in-database factorised computation. We show how the singular-value decomposition, the Moore-Penrose inverse, and the solution to linear least squares can be obtained from our approach, without materialising the data matrix $\boldsymbol{A}$.
- We investigate the computational complexity of **F-GS**. We show that **F-GS** has data complexity $\mathcal{O}(|\mathbf{D}|^{1+\max(fhtw(Q),2)})$ where $fhtw(Q)$ is the fractional hypertree width. In contrast, the data complexity of conventional approaches, which rely on the materialised join result, is $\mathcal{O}(|\mathbf{D}|^{\rho^*(Q)+2})$, where $\rho^*(Q)$ is the fractional edge cover number. The two measures satisfy $1 \leq fhtw(Q) \leq \rho^*(Q) \leq$ number of relations in Q. Note that the gap between these two measures can be significant for certain classes of queries commonly seen in learning tasks, e.g. acyclic queries for which $fhtw(Q)$ is 1 while $\rho^*(Q)$ can be up to $|Q|$.
- We show how to use **F-GS** to solve linear least squares and thus train linear regression models. This is the first in-database solution capable of computing the exact least square estimator over normalised data without materialising the design matrix $\boldsymbol{A}$. By extension, exact linear regression can be performed on large databases on which conventional software will take much more time or even run out of memory.

- We provide a C++ implementation of **F-GS** which relies on an (ideally) factorised engine to compute a set of related aggregate queries over $Q(\mathbf{D})$. In this dissertation the recent **AC/DC** [8] engine was used, however other engines could be plugged in easily. For example, recent development on covers, a sparse lossless representation of relational databases, could lead to an alternative aggregate engine [9].
- We report on extensive benchmarking with **F-GS** and two popular open-source competitors. Data matrices with different characteristics are defined using three different datasets, of which two real-world ones and one synthetic. We show that **F-GS** can outperform these competitors by more than 20× on a real-world dataset. Furthermore, in many cases the competitors are unable to load or one-hot encode the data altogether without running out of memory.

Matrix decompositions are an important part of linear algebra. This dissertation introduces the first matrix decomposition to factorised databases. By extension, opening up a line of research into potentially useful matrix decompositions in an in-database setting.

1.2 Outline

The structure of the remaining chapters in this dissertation is as follows:

Chapter 2 introduces notation and background theory on linear least squares, one-hot encoding categorical data, the QR decomposition, and factorised databases.

Chapter 3 presents the main theoretical results of this dissertation. First, an algebraic derivation of **F-GS** from the Gram-Schmidt process is given. The results are translated into an algorithm, for which the runtime complexity is analysed. The chapter ends with different applications of the factorised QR decompositions.

Chapter 4 gives details on the C++ implementation of the proposed system. The chapter outlines the data structures and optimisations used in **F-GS**. In particular, details regarding the sparse encoding of categorical data and the parallelisation of **F-GS** are described.

Chapter 5 describes the experimental setup and datasets. The performance of **F-GS** is compared against the systems **Py** and **R** which rely on the industry standard (LAPACK) for QR decomposition. Moreover, the chapter includes an analysis of **F-GS** through considering variants and a variety of problem instances.

Chapter 6 outlines related work on in-database analytics, in particular factorised learning and in-database linear algebra.

Chapter 7 summarises the outcomes of this dissertation and discusses interesting directions for future work in light of this dissertation.

Chapter 2

Preliminaries

This chapter outlines the prerequisite theory behind the dissertation, and describes the notions that are required to understand the proceeding chapters. It provides a brief description of the linear least squares problem. Next, a more detailed description of the QR decomposition is given. In particular, the Gram-Schmidt process is outlined and demonstrated with an example. It then explains the one-hot encoding, which is typically used to include categorical variables in a learning task. Finally, factorised databases are outlined, with a focus on concepts that are directly relevant to this dissertation.

For an exhaustive description of linear regression see [10]. Similarly, a comprehensive overview of factorised databases can be found in [11].

2.1 Notation and definitions

This section introduces important definitions and notation used in the remainder of this dissertation. More specialised notation may be introduced in the relevant sections.

2.1.1 Basics

$[k]$	The set of integers $\{1, 2, \ldots, k\}$
a	A scalar
$\boldsymbol{a}$	A vector
$\boldsymbol{A}$	A matrix
$\mathbf{0}$	The zero vector or matrix (clear from context)
$\boldsymbol{I}$	Identity matrix with implicit (from context) dimensions
$\mathbf{D}$	A database consisting of normalised relations
$\mathcal{O}$	Big O notation for data complexity

2.1.2 Linear Algebra

$\|\boldsymbol{a}\|$	The L^2 (Euclidean) norm of $\boldsymbol{a}$
$\langle \boldsymbol{u}, \boldsymbol{v} \rangle$	The dot product; inner product of Euclidean spaces.
$\boldsymbol{A}^\top$	The transpose of a matrix $\boldsymbol{A}$

2.1.2.1 Orthogonality

Orthogonality of vectors is the generalisation of perpendicularity to vectors of any dimension. A set of vectors $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n\}$ is (pairwise) orthogonal if for each pairs of vectors their inner product is zero.

An *orthogonal matrix* is a square matrix with orthogonal normalised (i.e. norm of 1) vectors as columns and rows, i.e.:

$$\boldsymbol{Q}^\top = \boldsymbol{Q}^{-1} \iff \boldsymbol{Q}^\top \boldsymbol{Q} = \boldsymbol{Q}\boldsymbol{Q}^\top = \boldsymbol{I}$$

In this dissertation we loosen the definition to include non-square semi-orthogonal matrices. A matrix with more rows than columns is semi-orthogonal, if the columns are orthonormal vectors. Equivalently, we consider a matrix to be (semi-)orthogonal if:

$$\boldsymbol{Q}^\top \boldsymbol{Q} = \boldsymbol{I} \text{ or } \boldsymbol{Q}\boldsymbol{Q}^\top = \boldsymbol{I}$$

2.1.2.2 Linear Dependence and Rank

A set of vectors $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n\}$ is *linearly dependent* if one of the vectors can be expressed as a linear combination of other vectors in the set. Equivalently, if there exist non-trivial coefficients $\boldsymbol{\lambda} = (\lambda_1, \lambda_2, \ldots, \lambda_n)$ such that:

$$\lambda_1 \boldsymbol{v}_1 + \lambda_2 \boldsymbol{v}_2 + \cdots + \lambda_n \boldsymbol{v}_n = \boldsymbol{0}$$

Conversely, the set of vectors is *linearly independent* if the only satisfying assignment is $\boldsymbol{\lambda} = \boldsymbol{0}$.

The *rank* of a matrix $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ is the dimension of the vector space spanned by the columns (or equivalently rows) of $\boldsymbol{A}$. A matrix is (or has) *full-rank* if

$$\operatorname{rank}(\boldsymbol{A}) = \min(m, n)$$

Equivalently, either all rows or all columns of $\boldsymbol{A}$ must be linearly independent. Otherwise, the matrix is said to be *rank deficient*.

2.1.3 Functions

A (partial) *function* or *map* f maps elements from a set X (*domain*) to a single element of a set Y (*range*). In this dissertation, we include *partial* functions, i.e. for all $x \in X$ either $f(x) \in Y$ or $f(x)$ is undefined.

For the purpose of this dissertation, it is useful to consider a function f as a set M of pairs $([\boldsymbol{x}], y)$ with key $\boldsymbol{x}$ and payload $y \in \mathbb{R}$, such that $f(\boldsymbol{x}) = y$. Therefore, the following representation is used for a function M:

$$M = \big\{([\boldsymbol{x}_1], \phi_1), ([\boldsymbol{x}_2], \phi_2), \ldots, ([\boldsymbol{x}_k], \phi_k)\big\}$$

2.2 Linear Least Squares

The linear least squares problem arises naturally in many different fields, and is most commonly used to perform linear regression. Consider a data set with m observations and N features:

$$\{y^{(i)}, x_1^{(i)}, \ldots, x_N^{(i)}\}_{i \in [m]}$$

The linear regression model assumes a linear relationship between the dependent variable or label (y_i) and the features ($x_{i1}, \ldots, x_{iN}$). Moreover, an error variable ε_i is included to capture the noise or error of each observation, such that:

$$y_i = \beta_0 + \beta_1 x_1^{(i)} + \cdots + \beta_n x_N^{(i)} + \varepsilon_i$$

For notational convenience, it is common to include $x_0^{(i)} := 1$ such:

$$\boldsymbol{y} = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{pmatrix}, \quad \boldsymbol{\beta} = \begin{pmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_N \end{pmatrix}, \quad \boldsymbol{\varepsilon} = \begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_m \end{pmatrix}$$

$$\boldsymbol{X} = \begin{pmatrix} \boldsymbol{x}_0 & \boldsymbol{x}_1 & \cdots & \boldsymbol{x}_m \end{pmatrix} = \begin{pmatrix} 1 & x_1^{(1)} & \ldots & x_N^{(1)} \\ 1 & x_1^{(2)} & \ldots & x_N^{(2)} \\ \vdots & \vdots & \cdots & \vdots \\ 1 & x_1^{(m)} & \ldots & x_N^{(m)} \end{pmatrix},$$

We can succinctly write the model using matrix notation as follows:

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon}$$

The goal of linear regression is to fit (or estimate) the parameter vector $\boldsymbol{\beta}$. Even though multiple estimation methods exist and are used in practice, we only consider the least squares principle. Figure 2.1 shows an illustrative example of least squares. Least squares sets out to minimise the sum of squared residuals (errors). The least squares estimator $\hat{\boldsymbol{\beta}}$ satisfies:

$$\hat{\boldsymbol{\beta}} = \arg\min_{\boldsymbol{\beta}} \sum_{i=1}^{m} {\varepsilon_i}^2 = \arg\min_{\boldsymbol{\beta}} \sum_{i=1}^{m} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}\|^2 \tag{2.1}$$

More generally; given an *overdetermined* (i.e. more equations than unknowns) system of equations $\boldsymbol{A}\boldsymbol{v} = \boldsymbol{b}$, the linear least squares problem is to find the solution which minimises the residuals.

$$\|\boldsymbol{b} - \boldsymbol{A}\boldsymbol{v}\|^2$$

It is well known that there exists a closed-form solution to equation 2.1, assuming that $\boldsymbol{A}$ is full-rank. The solution is obtained by solving the *Normal equations* which can be obtained by differentiating the objective with respect to $\boldsymbol{v}$.

$$(\boldsymbol{A}^\top \boldsymbol{A})\hat{\boldsymbol{v}} = \boldsymbol{A}^\top \boldsymbol{b} \iff \hat{\boldsymbol{v}} = (\boldsymbol{A}^\top \boldsymbol{A})^{-1} \boldsymbol{A}^\top \boldsymbol{b}$$

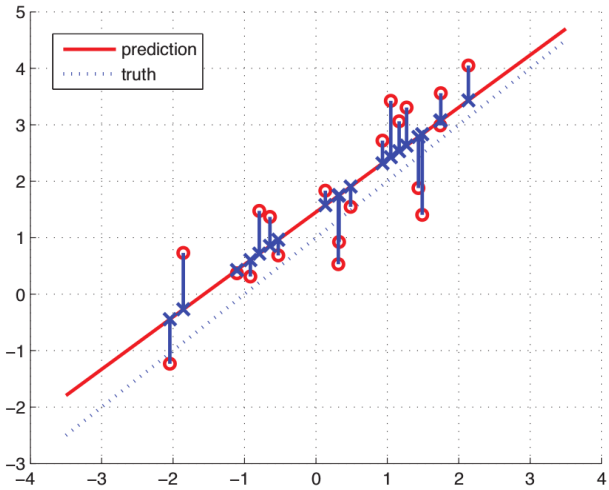


Figure 2.1: Example of a linear least squares regression instance with one feature. The regression line (shown in red) minimises the sum of squared errors (shown in blue), to best fit the data (shown as red circles). Figure taken from [10].

2.3 QR Decomposition

Definition 2.1. The QR decomposition of a matrix $\boldsymbol{A}$ is a decomposition of $\boldsymbol{A}$ into an orthogonal matrix $\boldsymbol{Q}_A$ (i.e. $\boldsymbol{Q}_A^\top \boldsymbol{Q}_A = \boldsymbol{I}$) and an upper triangular matrix $\boldsymbol{R}_A$ such that $\boldsymbol{A} = \boldsymbol{Q}_A \boldsymbol{R}_A$.

The best known use of the QR decomposition is to solve the linear least squares problem. The main advantage of using the QR decomposition to solve an instance of linear least squares, as opposed to a more direct approach (i.e. the Normal equations method), is numeric stability. Other uses include the (practical) QR algorithm and computing the singular-value decomposition of a matrix.

Any real $m \times n$ matrix $\boldsymbol{A}$ with $m \geq n$ admits a QR decomposition such that $\boldsymbol{Q}_A \in \mathbb{R}^{m \times m}$ and $\boldsymbol{R}_A \in \mathbb{R}^{m \times n}$. Since $\boldsymbol{R}_A$ is rectangular and upper triangular, the bottom $m - n$ rows consist entirely of zeroes. Therefore, it is useful to partition $\boldsymbol{Q}_A$ and $\boldsymbol{R}_A$.

$$\boldsymbol{A} = \boldsymbol{Q}_A \boldsymbol{R}_A = \boldsymbol{Q}_A \begin{bmatrix} \boldsymbol{R} \\ \boldsymbol{0} \end{bmatrix} = \begin{bmatrix} \boldsymbol{Q} & \boldsymbol{Q}' \end{bmatrix} \begin{bmatrix} \boldsymbol{R} \\ \boldsymbol{0} \end{bmatrix} = \boldsymbol{Q}\boldsymbol{R}$$

We have $m \times n$ matrix $\boldsymbol{Q}$, $n \times n$ matrix $\boldsymbol{R}$ and $(m - n) \times n$ matrix $\boldsymbol{Q}'$. The decomposition $\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{R}$ is known as a *thin* QR decomposition.

Multiple methods for computing the QR decomposition of a matrix exist; most notably the Gram-Schmidt process, Householder transformations, and Givens rotations.

2.3.1 Gram-Schmidt process

The Gram-Schmidt process is used to orthonormalise a set of vectors, and can be applied to the columns of a matrix to compute its QR decomposition. Gram-Schmidt computes the thin QR decomposition. Consider the matrix $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ with n linearly independent columns (i.e. full-rank) and $m \geq n$.

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{a}_1 & \boldsymbol{a}_2 & \cdots & \boldsymbol{a}_n \end{bmatrix}$$

The Gram-Schmidt process is defined as follows:

$$\begin{aligned}
\boldsymbol{u}_1 &= \boldsymbol{a}_1 && \boldsymbol{e}_1 = \frac{\boldsymbol{u}_1}{\|\boldsymbol{u}_1\|} \\
\boldsymbol{u}_2 &= \boldsymbol{a}_2 - \frac{\langle \boldsymbol{u}_1,\ \boldsymbol{a}_2 \rangle}{\langle \boldsymbol{u}_1,\ \boldsymbol{u}_1 \rangle} \boldsymbol{u}_1 && \boldsymbol{e}_2 = \frac{\boldsymbol{u}_2}{\|\boldsymbol{u}_2\|} \\
\boldsymbol{u}_k &= \boldsymbol{a}_k - \frac{\langle \boldsymbol{u}_1,\ \boldsymbol{a}_k \rangle}{\langle \boldsymbol{u}_1,\ \boldsymbol{u}_1 \rangle} \boldsymbol{u}_1 - \cdots - \frac{\langle \boldsymbol{u}_{k-1},\ \boldsymbol{a}_k \rangle}{\langle \boldsymbol{u}_{k-1},\ \boldsymbol{u}_{k-1} \rangle} \boldsymbol{u}_{k-1} && \boldsymbol{e}_k = \frac{\boldsymbol{u}_k}{\|\boldsymbol{u}_k\|} \\
&= \boldsymbol{a}_k - \sum_{i=1}^{k-1} \frac{\langle \boldsymbol{u}_i,\ \boldsymbol{a}_k \rangle}{\langle \boldsymbol{u}_i,\ \boldsymbol{u}_i \rangle} \boldsymbol{u}_i
\end{aligned}$$

A QR decomposition of $\boldsymbol{A}$ is then given by:

$$\boldsymbol{Q} = \begin{bmatrix} \boldsymbol{e}_1 & \boldsymbol{e}_2 & \cdots & \boldsymbol{e}_n \end{bmatrix}$$

$$\boldsymbol{R} = \begin{pmatrix} \langle \boldsymbol{e}_1,\ \boldsymbol{a}_1 \rangle & \langle \boldsymbol{e}_1,\ \boldsymbol{a}_2 \rangle & \cdots & \langle \boldsymbol{e}_1,\ \boldsymbol{a}_n \rangle \\ & \langle \boldsymbol{e}_2,\ \boldsymbol{a}_2 \rangle & \cdots & \langle \boldsymbol{e}_2,\ \boldsymbol{a}_n \rangle \\ & & \ddots & \vdots \\ \mathbf{0} & & & \langle \boldsymbol{e}_n,\ \boldsymbol{a}_n \rangle \end{pmatrix}$$

Computing a QR decomposition using the Gram-Schmidt process is appealing because of the simplicity of implementation and the intuitive geometric interpretation. The vector $\boldsymbol{u}_k$ is computed by projecting $\boldsymbol{a}_k$ orthogonally onto $\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_{k-1}$ and defining $\boldsymbol{u}_k$ as the difference between $\boldsymbol{a}_k$ and the projections. Next, $\boldsymbol{e}_k$ is obtained by normalising $\boldsymbol{u}_k$.

Example. To further illustrate the Gram-Schmidt QR decomposition, consider the following 3×3 matrix $\boldsymbol{A}$:

$$\boldsymbol{A} = \begin{pmatrix} 8 & 30 & 5 \\ 4 & 0 & 7 \\ -8 & -15 & 22 \end{pmatrix}$$

First, the Gram-Schmidt process is applied to find $\boldsymbol{u}_1$, $\boldsymbol{u}_2$ and $\boldsymbol{u}_3$.

$$\begin{aligned}
\boldsymbol{u}_1 &= \boldsymbol{a}_1 = \begin{pmatrix} 8 & 4 & -8 \end{pmatrix}^\top \\
\boldsymbol{u}_2 &= \boldsymbol{a}_2 - \frac{\langle \boldsymbol{u}_1,\ \boldsymbol{a}_2 \rangle}{\langle \boldsymbol{u}_1,\ \boldsymbol{u}_1 \rangle} \boldsymbol{u}_1 \\
&= \begin{pmatrix} 30 & 0 & -15 \end{pmatrix}^\top - \frac{\left\langle \begin{pmatrix} 8 & 4 & -8 \end{pmatrix},\ \begin{pmatrix} 30 & 0 & -15 \end{pmatrix} \right\rangle}{\left\langle \begin{pmatrix} 8 & 4 & -8 \end{pmatrix},\ \begin{pmatrix} 8 & 4 & -8 \end{pmatrix} \right\rangle} \begin{pmatrix} 8 & 4 & -8 \end{pmatrix}^\top \\
&= \begin{pmatrix} 30 & 0 & -15 \end{pmatrix}^\top - \frac{8 \cdot 30 + 8 \cdot 15}{8^2 + 4^2 + (-8)^2} \begin{pmatrix} 8 & 4 & -8 \end{pmatrix}^\top \\
&= \begin{pmatrix} 30 & 0 & -15 \end{pmatrix}^\top - \frac{5}{2} \begin{pmatrix} 8 & 4 & -8 \end{pmatrix}^\top \\
&= \begin{pmatrix} 10 & -10 & 5 \end{pmatrix}^\top
\end{aligned}$$

$$
\begin{aligned}
\boldsymbol{u}_3 &= \boldsymbol{a}_3 - \frac{\langle \boldsymbol{u}_1,\ \boldsymbol{a}_3 \rangle}{\langle \boldsymbol{u}_1,\ \boldsymbol{u}_1 \rangle}\boldsymbol{u}_1 - \frac{\langle \boldsymbol{u}_2,\ \boldsymbol{a}_3 \rangle}{\langle \boldsymbol{u}_2,\ \boldsymbol{u}_2 \rangle}\boldsymbol{u}_2 \\
&= \boldsymbol{a}_3 - \frac{\langle \begin{pmatrix} 8 & 4 & -8 \end{pmatrix},\ \begin{pmatrix} 5 & 7 & 22 \end{pmatrix} \rangle}{\langle \begin{pmatrix} 8 & 4 & -8 \end{pmatrix},\ \begin{pmatrix} 8 & 4 & -8 \end{pmatrix} \rangle}\boldsymbol{u}_1 - \frac{\langle \begin{pmatrix} 10 & -10 & 5 \end{pmatrix},\ \begin{pmatrix} 5 & 7 & 22 \end{pmatrix} \rangle}{\langle \begin{pmatrix} 10 & -10 & 5 \end{pmatrix},\ \begin{pmatrix} 10 & -10 & 5 \end{pmatrix} \rangle}\boldsymbol{u}_2 \\
&= \boldsymbol{a}_3 - \frac{8 \cdot 5 + 4 \cdot 7 - 8 \cdot 22}{8^2 + 4^2 + (-8)^2}\boldsymbol{u}_1 - \frac{10 \cdot 5 - 10 \cdot 7 + 5 \cdot 22}{10^2 + (-10)^2 + 5^2}\boldsymbol{u}_2 \\
&= \begin{pmatrix} 5 & 7 & 22 \end{pmatrix}^\top + \frac{3}{4}\begin{pmatrix} 8 & 4 & -8 \end{pmatrix}^\top - \frac{2}{5}\begin{pmatrix} 10 & -10 & 5 \end{pmatrix}^\top \\
&= \begin{pmatrix} 7 & 14 & 14 \end{pmatrix}^\top
\end{aligned}
$$

Next, $\boldsymbol{Q}$ is obtained by normalising the vectors $\boldsymbol{u}_1$, $\boldsymbol{u}_2$, and $\boldsymbol{u}_3$.

$$
\begin{aligned}
\boldsymbol{Q} &= \begin{bmatrix} \frac{\boldsymbol{u}_1}{\|\boldsymbol{u}_1\|} & \frac{\boldsymbol{u}_2}{\|\boldsymbol{u}_2\|} & \frac{\boldsymbol{u}_3}{\|\boldsymbol{u}_3\|} \end{bmatrix} \\
&= \begin{pmatrix} \frac{8}{12} & \frac{10}{15} & \frac{7}{21} \\ \frac{4}{12} & -\frac{10}{15} & \frac{14}{21} \\ -\frac{8}{12} & \frac{5}{15} & \frac{14}{21} \end{pmatrix} = \begin{pmatrix} \frac{2}{3} & \frac{2}{3} & \frac{1}{3} \\ \frac{1}{3} & -\frac{2}{3} & \frac{2}{3} \\ -\frac{2}{3} & \frac{1}{3} & \frac{2}{3} \end{pmatrix}
\end{aligned}
$$

Finally, the matrix $\boldsymbol{R}$ is obtained.

$$
\begin{aligned}
\boldsymbol{R} &= \begin{pmatrix} \frac{\langle \boldsymbol{u}_1,\, \boldsymbol{a}_1 \rangle}{\|\boldsymbol{u}_1\|} & \frac{\langle \boldsymbol{u}_1,\, \boldsymbol{a}_2 \rangle}{\|\boldsymbol{u}_1\|} & \frac{\langle \boldsymbol{u}_1,\, \boldsymbol{a}_3 \rangle}{\|\boldsymbol{u}_1\|} \\ 0 & \frac{\langle \boldsymbol{u}_2,\, \boldsymbol{a}_2 \rangle}{\|\boldsymbol{u}_2\|} & \frac{\langle \boldsymbol{u}_2,\, \boldsymbol{a}_3 \rangle}{\|\boldsymbol{u}_2\|} \\ 0 & 0 & \frac{\langle \boldsymbol{u}_3,\, \boldsymbol{a}_3 \rangle}{\|\boldsymbol{u}_3\|} \end{pmatrix} \\
&= \begin{pmatrix} 12 & 30 & -9 \\ 0 & 15 & 6 \\ 0 & 0 & 21 \end{pmatrix}
\end{aligned}
$$

This shows how the Gram-Schmidt process can be used to compute a QR decomposition of a matrix $\boldsymbol{A}$. However, the Gram-Schmidt process is inherently numerically unstable due to a loss of orthogonality as a result of (floating-point) rounding errors. In practice a minor modification is sometimes used, resulting in a more stable version known as *modified Gram-Schmidt* [12].

2.3.1.1 Modified Gram Schmidt

Instead of computing $\boldsymbol{u}_k$ as a sum of projections, an iterative approach is used.

$$
\begin{aligned}
\boldsymbol{u}_k^{(1)} &= \boldsymbol{a}_k - \frac{\langle \boldsymbol{u}_1,\ \boldsymbol{a}_k \rangle}{\langle \boldsymbol{u}_1,\ \boldsymbol{u}_1 \rangle}\boldsymbol{u}_1 \\
\boldsymbol{u}_k^{(2)} &= \boldsymbol{u}_k^{(1)} - \frac{\langle \boldsymbol{u}_2,\ \boldsymbol{u}_k^{(1)} \rangle}{\langle \boldsymbol{u}_2,\ \boldsymbol{u}_2 \rangle}\boldsymbol{u}_k^{(1)} \\
&\vdots \\
\boldsymbol{u}_k^{(i)} &= \boldsymbol{u}_k^{(i-1)} - \frac{\langle \boldsymbol{u}_i,\ \boldsymbol{u}_k^{(i-1)} \rangle}{\langle \boldsymbol{u}_i,\ \boldsymbol{u}_i \rangle}\boldsymbol{u}_k^{(i-1)} \\
\boldsymbol{u}_k &= \boldsymbol{u}_k^{(k)}
\end{aligned}
$$

At each step $\boldsymbol{u}_k^{(i)}$ is orthogonalised against the previous step $\boldsymbol{u}_k^{(i-1)}$, including any error that was introduced in the computation of $\boldsymbol{u}_k^{(i-1)}$. Noteworthy is that (classical) Gram-Schmidt and modified Gram-Schmidt are mathematically equivalent; that is in exact arithmetic.

2.3.2 Solving Linear Least Squares

Consider the system of equations $\boldsymbol{Ax} = \boldsymbol{b}$ with $\boldsymbol{A} \in \mathbb{R}^{m \times n}$, $\boldsymbol{x} \in \mathbb{R}^n$ and $\boldsymbol{b} \in \mathbb{R}^m$. We want the solve the linear least squares such that $\hat{\boldsymbol{x}}$ minimises:

$$\|\boldsymbol{b} - \boldsymbol{Ab}\|^2$$

The closed-form solution is obtained by solving the Normal equations.

$$(\boldsymbol{A}^\top \boldsymbol{A})\hat{\boldsymbol{x}} = \boldsymbol{A}^\top \boldsymbol{b}$$

The orthogonality of the $\boldsymbol{Q}$ matrix in the QR decomposition of $\boldsymbol{A} = \boldsymbol{QR}$ can be exploited to efficiently solve for $\hat{\boldsymbol{x}}$.

$$\begin{aligned}
(\boldsymbol{A}^\top \boldsymbol{A})\hat{\boldsymbol{x}} &= \boldsymbol{A}^\top \boldsymbol{b} && \iff \\
((\boldsymbol{QR})^\top \boldsymbol{QR})\hat{\boldsymbol{x}} &= (\boldsymbol{QR})^\top \boldsymbol{b} && \iff \\
(\boldsymbol{R}^\top \boldsymbol{Q}^\top \boldsymbol{QR})\hat{\boldsymbol{x}} &= \boldsymbol{R}^\top \boldsymbol{Q}^\top \boldsymbol{b} && \iff \\
\boldsymbol{R}^\top \boldsymbol{R}\hat{\boldsymbol{x}} &= \boldsymbol{R}^\top \boldsymbol{Q}^\top \boldsymbol{b} && \iff \\
\boldsymbol{R}\hat{\boldsymbol{x}} &= \boldsymbol{Q}^\top \boldsymbol{b}
\end{aligned}$$

Above expression is rewritten by first computing $\boldsymbol{d} = \boldsymbol{Q}^\top \boldsymbol{b}$ resulting in the upper triangular system $\boldsymbol{R}\hat{\boldsymbol{x}} = \boldsymbol{d}$. Consider the system written as linear equations in reverse order:

$$\begin{array}{ccccccccc}
r_{n,n} \cdot x_n & & & & & & & = & d_n \\
r_{n-1,n} \cdot x_n & + & r_{n-1,n-1} \cdot x_{n-1} & & & & & = & d_{n-1} \\
\vdots & & \vdots & & \ddots & & & & \vdots \\
r_{1,n} \cdot x_n & + & r_{n-1,n-1} \cdot x_{n-1} & + & \ldots & + & r_{1,1} \cdot x_1 & = & d_1
\end{array}$$

By considering the equations in reverse order, we can solve $\hat{x}_n$ at the first step. By substituting $\hat{x}_n$ in the next step, we can solve for the only free variable $\hat{x}_{n-1}$.

$$\begin{aligned}
\hat{x}_n &= \frac{d_n}{r_{n,n}} \\
\hat{x}_{n-1} &= \frac{d_{n-1} - r_{n-1,n}\hat{x}_n}{r_{n-1,n-1}} \\
\hat{x}_k &= \frac{1}{r_{k,k}} \cdot \left(d_k - \sum_{i=k+1}^{n} r_{k,i}\hat{x}_i\right)
\end{aligned}$$

This procedure avoids inverting $\boldsymbol{R}$ to more efficiently solve an upper triangular system, and is known as *backward substitution* because of the reversed order of the equations. The time complexity of backward substitution on an $n \times n$ matrix is $O(n^2)$.

2.4 One-Hot Encoding Categorical Variables

A categorical variable refers to an attribute with a fixed limited number of possible values, assigning the entry to one specific nominal category (or group). Examples of categorical variables relating to a person include blood type, nationality, smoker, and gender. The QR decomposition and many machine learning algorithms require the input data to be numeric and therefore require categorical data to be encoded. The encoding used is important for the interpretation and treatment of categorical data.

Example. The variable "smoker" may be (compactly) stored as '0 = unknown', '1 = yes' and '2 = no'. However, a ML model may assume a natural ordering between the categories or incorporate the non-smoker category as 'twice' the smoker category.

Consequently, the *one-hot encoding* can be used to incorporate indicator (or dummy) features for the categories. At its core, one-hot coding extends the dataset by using one boolean (indicator) feature for each distinct category. The intuition is that the boolean feature indicates whether the entry belongs to that specific category.

In practice, this approach may result in a problem known as the *dummy variable trap*. The trap is best explained by a simple example.

Example. Consider a linear model with a single boolean variable 'verified', encoded as two boolean features $x_1^{(i)}$ (yes) and $x_2^{(i)}$ (no), and the intercept $x_0^{(i)}$.

$$y^{(i)} = \beta_0 x_0^{(i)} + \beta_1 x_1^{(i)} + \beta_2 x_2^{(i)}$$

Clearly, every entry must be either verified or unverified, hence:

$$\forall_{i \in [m]} : x_1^{(i)} + x_2^{(i)} = 1 = x_0^{(i)}$$

The result is a design matrix with perfect multicollinearity, which means that $\boldsymbol{X}$ is not full-rank and the Gram-Schmidt process cannot be applied. In fact, for rank-deficient $\boldsymbol{X}$ no unique solution to the least squares problem exists.

The dummy variable trap occurs whenever there is more than one categorical variable or one categorical variable and a constant feature (e.g. intercept in linear regression). The dummy variable trap can be avoided by removing one indicator feature from each categorical variable. Moreover, we ensure that a constant feature is included to retain the same rank as before dropping any features.

Within the context of a learning model, dropping one indicator feature per variable does not reduce the expressive or predictive power of the model. Instead, entries belonging to the excluded group are considered to be the "base" group and represented by the constant (intercept) term. The learned parameters for all other (included) groups are then relative to this base group.

Example. We end this section with a comparison of the integer and one-hot encoding. Table 2.2 shows both encodings for a small relation containing blood type information.

Entry	Blood type	Group	A	B	O
1	AB	1	0	0	0
2	A	2	1	0	0
3	A	2	1	0	0
4	B	3	0	1	0
5	O	4	0	0	1
6	B	3	0	1	0

Table 2.2: Example of a table containing blood type (categorical) information. The third column (group) represents the integer representation. The last three columns show the corresponding one-hot encoding, with 'AB' being excluded.

2.5 Factorised Databases

Relational databases store relations as tables consisting of records (i.e. rows) sharing the same attributes (i.e. columns). To avoid redundancy, a database is split into *normalised* relations which can be related to each other via join keys. It is common practice to materialise the join result of two or more relations on common attributes (i.e. keys), such as shown in Figure 2.3. The resulting *listing representation* contains a high degree of redundancy, which the normalised relations set out to eliminate. This data redundancy will lead to corresponding processing overhead, e.g. for computing aggregates over the join result.

Orders

Cust	Day	Menu
Mark	Mon	Regular
Alice	Mon	Regular
John	Mon	Vegetar.
John	Sun	Vegetar.

Menus

Menu	Item
Regular	Burger
Regular	Fries
Regular	Drink
Vegetar.	Salad
Vegetar.	Fries
Vegetar.	Drink

Preparation

Item	Time
Burger	8
Salad	6
Fries	4
Drink	1

(a) Normalised

Orders ⋈ Menus ⋈ Preparation

Cust	Day	Menu	Item	Time
Mark	Mon	Regular	Burger	8
Mark	Mon	Regular	Fries	4
Mark	Mon	Regular	Drink	1
Alice	Mon	Regular	Burger	8
Alice	Mon	Regular	Fries	4
Alice	Mon	Regular	Drink	1
John	Mon	Vegetar.	Salad	6
John	Mon	Vegetar.	Fries	4
John	Mon	Vegetar.	Drink	1
John	Sun	Vegetar.	Salad	6
John	Sun	Vegetar.	Fries	4
John	Sun	Vegetar.	Drink	1

(b) Natural Join

Figure 2.3: Example of a small relational database, with colouring to emphasise redundancy.

Alternatively, *factorised representations* are succinct lossless representations for arbitrary join queries over relational databases. They exploit properties of relational algebra, in particular the distributivity of the Cartesian product over union. More importantly, they led to the development of factorised aggregate computation engines [13][8]. Consequently, the computational and space complexity for solving *Functional Aggregate Queries* (FAQs), a generalisation of join queries, has been lowered for a range of join queries.

In general, there is no unique factorisation for a given join query Q over the database $\mathbf{D}$. A *variable ordering* defines the (nesting) structure of the factorised representation. A variable ordering is a partial ordering over the attributes, which

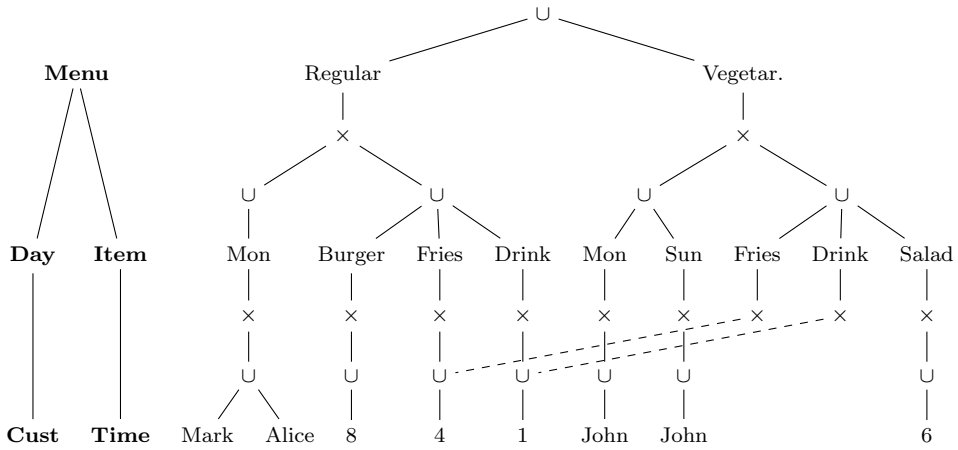


Figure 2.4: One possible factorisation of the join result (right) and the corresponding variable order (left). Dotted lines denote reuse of cached symbols.

is obtained from static analysis of Q and the schematic structure of $\mathbf{D}$. One possible factorisation is shown in Figure 2.4, with the variable ordering on the left.

The size of a factorisation depends on which variable ordering is used to dictate its structure. Even though the size of different factorisations may vary greatly, the asymptotically optimal sizes are well understood. Let $|\mathbf{D}|$ denote the number of tuples in the database $\mathbf{D}$.

Proposition 2.2 ([11])**.** *Given a feature extraction join query Q, with continuous features only, then for every database $\mathbf{D}$, the join result $Q(\mathbf{D})$ admits:*

- *a listing representation of size $\mathcal{O}(|\mathbf{D}|^{\rho^*(Q)})$;* [14]
- *a factorised representation of size $\mathcal{O}(|\mathbf{D}|^{fhtw(Q)})$;* [15]

There are classes of databases for which above bounds are tight and worst-case optimal join algorithms to compute the join result in these representations [16][15].

The parameters used in Proposition 2.2 are the fractional edge cover number $\rho^*(Q)$ and the fractional hypertree width $fhtw(Q)$, satisfying $1 \leq fhtw(Q) \leq \rho^*(Q) \leq |Q|$ with $|Q|$ the number of relations in Q. The gap between $fhtw(Q)$ and $\rho^*(Q)$ can be as large as $|Q| - 1$, in which case the factorised join result can be computed exponentially faster than the listing representation (even when using worst-case optimal join algorithms!) [7]. For example, acyclic joins (e.g. path and hierarchical) have $fhtw(Q) = 1$ whereas $\rho^*(Q)$ can be as large as the number of relations $|Q|$.

2.5.1 Cofactor Matrix

A particularly useful and recurring use of FAQs is to compute the *cofactor matrix*.

$$\mathsf{Cofactor} = \boldsymbol{A}^\top \boldsymbol{A} \in \mathbb{R}^{N \times N}$$

The cofactor matrix is commonly used in applications of statistics and machine learning. Factorised computation of $\boldsymbol{A}^\top \boldsymbol{A}$ has previously been used in multiple in-database learning tasks, including linear and polynomial regression, factorisation

machines and principle component analysis [6]. Most importantly, all aggregates in Cofactor can be computed in a single pass over the factorised join.

$$\mathsf{Cofactor} = \begin{bmatrix} \langle \boldsymbol{a}_1,\ \boldsymbol{a}_1 \rangle & \langle \boldsymbol{a}_1,\ \boldsymbol{a}_2 \rangle & \cdots & \langle \boldsymbol{a}_1,\ \boldsymbol{a}_N \rangle \\ \langle \boldsymbol{a}_2,\ \boldsymbol{a}_1 \rangle & \langle \boldsymbol{a}_2,\ \boldsymbol{a}_2 \rangle & \cdots & \langle \boldsymbol{a}_2,\ \boldsymbol{a}_N \rangle \\ \langle \boldsymbol{a}_3,\ \boldsymbol{a}_1 \rangle & \langle \boldsymbol{a}_3,\ \boldsymbol{a}_2 \rangle & \cdots & \langle \boldsymbol{a}_3,\ \boldsymbol{a}_N \rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle \boldsymbol{a}_N,\ \boldsymbol{a}_1 \rangle & \langle \boldsymbol{a}_N,\ \boldsymbol{a}_2 \rangle & \cdots & \langle \boldsymbol{a}_N,\ \boldsymbol{a}_N \rangle \end{bmatrix} \tag{2.2}$$

A useful property of the cofactor matrix is symmetry, which can easily be shown.

$$(\boldsymbol{A}^\top \boldsymbol{A})^\top = \boldsymbol{A}^\top (\boldsymbol{A}^\top)^\top = \boldsymbol{A}^\top \boldsymbol{A}$$

It follows that only the entries in the upper-half (including the diagonal) need to be computed.

2.5.2 Sparse Encoding for Categorical Variables

In Section 2.4 the one-hot encoding, which is often used to incorporate categorical variables in a learning model, is described. However, one-hot encoding generally leads to significant redundancy, especially when working with large datasets with multiple categorical variables. As an example, consider a categorical column with m rows and p possible categories (i.e. groups). One-hot encoding results in pm values of which only m are ones, and the remaining $(p-1)m$ are zeroes. Computationally it requires processing potentially many zeroes, and more importantly, the increased memory usage may result in a procedure running out of memory.

In factorised learning, an alternative sparse encoding of the input data has been proposed [6]. Instead of increasing the number of columns, each entry is represented by a pair consisting of a key and a payload of 1. For notational consistency, continuous variables are similarly encoded. Figure 2.5 shows columns of both types in their corresponding encoding.

$$\mathrm{enc}(\boldsymbol{x}_a) = \begin{bmatrix} ([x_a^{(1)}],\ 1) \\ ([x_a^{(2)}],\ 1) \\ \vdots \\ ([x_a^{(k)}],\ 1) \end{bmatrix} \qquad \mathrm{enc}\,(\boldsymbol{x}_b) = \begin{bmatrix} ([],\ x_b^{(1)}) \\ ([],\ x_b^{(2)}) \\ \vdots \\ ([],\ x_b^{(k)}) \end{bmatrix}$$

(a) Categorical variable (b) Continuous variable

Figure 2.5: Examples of encoded columns with k entries. The payloads in blue.

This encoding may not seem useful yet, however, it allows us to concisely express the computation of cofactors. More importantly, this encoding avoids aggregating categorical variables together in the same way as one-hot encoding, without introducing the associated redundancy. Finally, even though (conceptually) we think

of the input data as encoded in this manner, in practice this encoding is applied on-the-fly where necessary during aggregate computation.

Figure 2.6 compares the sparse encoding to the one-hot encoding.

$$\begin{bmatrix} \underbrace{10\ldots0}_{p} & \cdots & \underbrace{10\ldots0}_{p} \\ \vdots & \ddots & \vdots \\ \underbrace{0\ldots01}_{p} & \cdots & \underbrace{0\ldots01}_{p} \end{bmatrix}$$

(a) onehot $(\boldsymbol{A})$

$$\begin{bmatrix} ([a_1^{(1)}],\, 1) & \cdots & ([a_n^{(1)}],\, 1) \\ \vdots & \ddots & \vdots \\ ([a_1^{(m)}],\, 1) & \cdots & ([a_n^{(m)}],\, 1) \end{bmatrix}$$

(b) enc $(\boldsymbol{A})$

Figure 2.6: Side-by-side comparison of the two encodings applied to an $m \times n$ matrix with n categorical variables each with p categories.

2.5.3 Sigma: Sparsely Including Categorical Cofactors

The Sigma matrix ($\boldsymbol{\Sigma}$) is introduced to incorporate the sparse categorical encoding into the cofactor matrix. The entries in $\boldsymbol{\Sigma}$ are functions instead of scalars. Alternatively, each entry can be considered a predicate which is nullary, unary or binary depending on the domains of the variables. Moreover, the dimensions of $\boldsymbol{\Sigma}$ are proportional to the number of variables n instead of the number of features N.

$$\boldsymbol{\Sigma} = \begin{bmatrix} \sigma_{1,1} & \cdots & \sigma_{1,n} \\ \vdots & \ddots & \vdots \\ \sigma_{1,n} & \cdots & \sigma_{n,n} \end{bmatrix}$$

In order to express the entries in $\boldsymbol{\Sigma}$ we introduce the sum and product over mappings. For the purpose of this dissertation it is sufficient to define the product over pairs.

$$([\boldsymbol{x}],\, \phi_x) \otimes ([\boldsymbol{y}],\, \phi_y) := \big([\boldsymbol{x}, \boldsymbol{y}],\, \phi_x \cdot \phi_y\big)$$

The slightly more generalised sum is defined over a mapping M and a pair.

$$M \oplus ([\boldsymbol{y}], \phi_y) := \begin{cases} \big(M \setminus \big\{([\boldsymbol{y}], \phi_x)\big\}\big) \cup \big\{([\boldsymbol{y}], \phi_x + \phi_y)\big\} & \text{if } \exists \phi_x \in \mathbb{R} : ([\boldsymbol{y}], \phi_x) \in M \\ M \cup \big\{([\boldsymbol{y}], \phi_y)\big\} & \text{otherwise} \end{cases}$$

Informally, the product ($\otimes$) merges two pairs by concatenating their keys and multiplying the payloads. The sum ($\oplus$) resembles a union; if the mapping M already contains the key of the summed pair, then the payload of the summed pair is added to the existing pair. If not, the pair is simply inserted into M.

Using these definitions, we can generalise the dot product to include mappings.

$$\begin{aligned} \sigma_{k,l} &= \langle \text{enc}\,(\boldsymbol{a}_k),\ \text{enc}\,(\boldsymbol{a}_l) \rangle \\ &= \bigoplus_{i \in [m]} \big[\, \text{enc}\,(a_k^{(i)}) \otimes \text{enc}\,(a_l^{(i)}) \big] \end{aligned}$$

For example, an entry for two continuous variables $\boldsymbol{a}_k, \boldsymbol{a}_l$ contains a singleton.

$$\sigma_{k,l} = \{([],\ \phi_{k,l})\}$$

Whereas an entry for a categorical variable $\boldsymbol{a}_k$ with p categories and a continuous variable $\boldsymbol{a}_l$ contains p entries.

$$\sigma_{k,l} = \{([k_1],\ \phi_{k_1,l}),\ ([k_2],\ \phi_{k_2,l}),\ \cdots,\ ([k_p],\ \phi_{k_p,l})\}$$

Example. To clarify the sparse encoding and dot product, consider the small table:

$$\text{enc}\left(\begin{array}{|ccc|}\hline \underline{\textbf{I}}\textbf{tem} & \underline{\textbf{C}}\textbf{ustomer} & \underline{\textbf{A}}\textbf{mount} \\ \hline A & X & 4 \\ C & Y & 1 \\ B & X & 1 \\ C & Y & 3 \\ \hline\end{array}\right) = \begin{array}{|ccc|}\hline \textbf{I} & \textbf{C} & \textbf{A} \\ \hline ([A],\ 1) & ([X],\ 1) & ([],\ 4) \\ ([C],\ 1) & ([Y],\ 1) & ([],\ 1) \\ ([B],\ 1) & ([X],\ 1) & ([],\ 1) \\ ([C],\ 1) & ([Y],\ 1) & ([],\ 3) \\ \hline\end{array}$$

We proceed to show how some of the entries in $\boldsymbol{\Sigma}$ can be computed.

$$\begin{aligned}
\sigma_{I,A} &= \langle \text{enc}\,(\boldsymbol{a}_I),\ \text{enc}\,(\boldsymbol{a}_A)\rangle \\
&= \bigoplus_{i\in[m]} \left(\text{enc}\,(a_I^{(i)}) \otimes \text{enc}\,(a_A^{(i)})\right) \\
&= \left(([A],\ 1) \otimes ([],\ 4)\right) \oplus \left(([C],\ 1) \otimes ([],\ 1)\right) \oplus \left(([B],\ 1) \otimes ([],\ 1)\right) \oplus \\
&\quad \left(([C],\ 1) \otimes ([],\ 3)\right) \\
&= ([A],\ 4) \oplus ([C],\ 1) \oplus ([B],\ 1) \oplus ([C],\ 3) \\
&= \{([A],\ 4),\ ([C],\ 1)\} \oplus ([B],\ 1) \oplus ([C],\ 3) \\
&= \{([A],\ 4),\ ([C],\ 1),\ ([B],\ 1)\} \oplus ([C],\ 3) \\
&= \{([A],\ 4),\ ([C],\ 4),\ ([B],\ 1)\} \\
\sigma_{A,A} &= \langle \text{enc}\,(\boldsymbol{a}_A),\ \text{enc}\,(\boldsymbol{a}_A)\rangle \\
&= \left(([],\ 4) \otimes ([],\ 4)\right) \oplus \left(([],\ 1) \otimes ([],\ 1)\right) \oplus \left(([],\ 1) \otimes ([],\ 1)\right) \oplus \left(([],\ 3) \otimes ([],\ 3)\right) \\
&= ([],\ 16) \oplus ([],\ 1) \oplus ([],\ 1) \oplus ([],\ 9) \\
&= \{([],\ 27)\} \\
\sigma_{I,C} &= \langle \text{enc}\,(\boldsymbol{a}_I),\ \text{enc}\,(\boldsymbol{a}_C)\rangle \\
&= \left(([A],1) \otimes ([X],1)\right) \oplus \left(([C],1) \otimes ([Y],1)\right) \oplus \left(([B],1) \otimes ([X],1)\right) \oplus \\
&\quad \left(([C],1) \otimes ([Y],1)\right) \\
&= ([A,X],\ 1) \oplus ([C,Y],\ 1) \oplus ([B,X],\ 1) \oplus ([[C,Y],\ 1) \\
&= \{([A,X],\ 1),\ ([C,Y],\ 2),\ ([B,X],\ 1)\}
\end{aligned}$$

Finally, we show how the aggregates in an entry (i.e. $\sigma_{k,l}$) can be expressed as FAQs. Consider two arbitrary keys x_k and x_l in the mappings of $\boldsymbol{a}_k$ and $\boldsymbol{a}_l$ (respectively).

- If x_k and x_l both have continuous domains:

$$\sigma_{k,l}() = \sum_{\forall j\in[n]:x_j} x_k \cdot x_l \cdot \mathbf{1}_{Q(x_1,\cdots x_n)}$$

- If x_k and x_l both have categorical domains:

$$\sigma_{k,l}(x_k, x_l) = \sum_{\forall j \in [n]: x_j}^{j \notin \{k,l\}} \mathbf{1}_{Q(x_1, \cdots x_n)}$$

- If x_k has a categorical domain, and x_l has a continuous domain:

$$\sigma_{k,l}(x_k) = \sum_{\forall j \in [n]: x_j}^{j \notin \{k\}} x_l \cdot \mathbf{1}_{Q(x_1, \cdots x_n)}$$

Chapter 3

Factorised Gram-Schmidt

This chapter describes *factorised Gram-Schmidt*, the theoretical result behind the main technical contribution of this dissertation. **F-GS** can be considered a two-layer factorisation of a QR decomposition computation. This chapter describes the first layer which is a decomposition of the Gram-Schmidt process into subproblems expressible as relation queries. The second layer refers to exploiting techniques from factorised databases to push these subproblems past the join of the input relations.

First, an outline of the factorised Gram-Schmidt process and its in-database setting are provided. Next, an algebraic rewrite of the Gram-Schmidt process to expressions in terms of the cofactors is given. The obtained equations are then translated into an algorithm stated in pseudocode. An analysis of the space and time complexity of this algorithm is provided. Finally, some applications of the factorised QR decomposition are stated. For simplicity, this chapter assumes that the cofactors are given as the square $N \times N$ matrix Cofactor.

Chapter 4 then describes the C++ implementation of **F-GS**. Moreover, the results of this chapter are extended to include the sparse encoding of $\mathbf{\Sigma}$.

3.1 Outline and Setting

This dissertation assumes the common in-database scenarios where join queries are used to retrieve a matrix. Consider a feature extraction query Q and database $\mathbf{D}$.

$$\boldsymbol{A} = Q(\mathbf{D})$$

This chapter assumes that $\boldsymbol{A}$ consists of numerical values which were obtained using any preferred encoding for categorical variables. Let N be the number of features and let m be the number of rows, such that $\boldsymbol{A} \in \mathbb{R}^{m \times N}$. Most importantly, in a database setting it is a reasonable assumption that there are many more rows than there are features (or columns); that is $m \gg N$.

The factorised Gram-Schmidt process computes a factorised QR decomposition of the matrix $\boldsymbol{A}$.

Definition 3.1. The *factorised QR decomposition* of a matrix $\boldsymbol{A}$ is of shape:

$$\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{R} = (\boldsymbol{A}\boldsymbol{C})\boldsymbol{R}$$

Factorised refers to the fact that $\boldsymbol{A}$ and $\boldsymbol{Q}$ are kept symbolically, whereas the $N \times N$ matrices $\boldsymbol{C}$ and $\boldsymbol{R}$ are materialised.

The goal of factorised Gram-Schmidt is to express $\boldsymbol{C}$ and $\boldsymbol{R}$ in terms of the cofactors of $\boldsymbol{A}$. Consequently, it provides a computation for the factorised QR decomposition which relies solely on Cofactor instead of the actual data.

Recall that the second layer of our approach factorises the computation of the cofactors to avoid materialising the join result $\boldsymbol{A}$. Therefore, all stages of **F-GS** can be performed without ever materialising $\boldsymbol{A}$.

3.2 Rewriting the Gram-Schmidt Process

Recall the Gram-Schmidt process outlined in Section 2.3.1.

$$\boldsymbol{u}_k = \boldsymbol{a}_k - \sum_{i=1}^{k-1} \frac{\langle \boldsymbol{u}_i,\ \boldsymbol{a}_k \rangle}{\langle \boldsymbol{u}_i,\ \boldsymbol{u}_i \rangle} \boldsymbol{u}_i \tag{3.1}$$

$$\boldsymbol{e}_k = \frac{\boldsymbol{u}_k}{\|\boldsymbol{u}_k\|} = \frac{\boldsymbol{u}_k}{\sqrt{\langle \boldsymbol{u}_k,\ \boldsymbol{u}_k \rangle}} \tag{3.2}$$

The key insight of the proposed system is that the inner products in equations 3.1 and 3.2 can be expressed in terms of sum-product expressions. Most importantly, these expressions are subject to factorised computation in the same way that the join query is. Specifically, we rewrite the Gram-Schmidt process such that:

$$\begin{bmatrix} \boldsymbol{u}_1 & \boldsymbol{u}_2 & \cdots & \boldsymbol{u}_N \end{bmatrix} = \boldsymbol{A}\boldsymbol{C}'$$

$$= \begin{bmatrix} \boldsymbol{a}_1 & \boldsymbol{a}_2 & \cdots & \boldsymbol{a}_N \end{bmatrix} \begin{pmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,N} \\ & c_{2,2} & \cdots & c_{2,N} \\ & & \ddots & \vdots \\ \mathbf{0} & & & c_{N,N} \end{pmatrix}$$

We set out to show that we can express $\boldsymbol{u}_k$ as a linear combination of the columns of $\boldsymbol{A}$.

Theorem 3.2. *For all $k \in [N]$, the orthogonalised vectors $\boldsymbol{u}_k$ can be expressed as:*

$$\boldsymbol{u}_k = \sum_{j \in [k]} c_{j,k} \boldsymbol{a}_j \tag{3.3}$$

Where for all $j \in [k]$: $c_{j,k}$ can be computed in terms of Cofactor.

Furthermore, we show that $\boldsymbol{C}$ and $\boldsymbol{R}$ can be expressed in terms of $\mathsf{Cofactor} = \boldsymbol{A}^\top \boldsymbol{A}$. That is, given $\mathsf{Cofactor} = \boldsymbol{A}^\top \boldsymbol{A}$ we can compute $\boldsymbol{C}$ and $\boldsymbol{R}$ in time independent of the number of rows m. Recall that Cofactor contains the inner products of the columns of $\boldsymbol{A}$.

$$\mathsf{Cofactor}[i, j] = \langle \boldsymbol{a}_i,\ \boldsymbol{a}_j \rangle$$

Proof of Theorem 3.2

First of all, observe that from equations 3.1 and 3.3 it is clear that $c_{i,i} = 1$ for all $i \in [N]$. We proceed to consider the base case $k = 2$:

$$\boldsymbol{u}_2 = \boldsymbol{a}_2 - \frac{\langle \boldsymbol{a}_1,\ \boldsymbol{a}_2 \rangle}{\langle \boldsymbol{a}_1,\ \boldsymbol{a}_1 \rangle} \boldsymbol{a}_1$$
$$c_{1,2} := -\frac{\langle \boldsymbol{a}_1,\ \boldsymbol{a}_2 \rangle}{\langle \boldsymbol{a}_1,\ \boldsymbol{a}_1 \rangle}$$

To generalise this, using proof by induction, assume that for some k it holds:

$$\forall\, t \in [k-1] \colon\ \boldsymbol{u}_t = \sum_{j \in [t]} c_{j,t} \boldsymbol{a}_j$$

We use the definition of $\boldsymbol{u}_k$ (equation 3.1) to find the necessary terms to rewrite $\boldsymbol{u}_k$.

$$\begin{aligned}
\boldsymbol{u}_k &= \boldsymbol{a}_k - \sum_{i=1}^{k-1} \frac{\langle \boldsymbol{u}_i,\ \boldsymbol{a}_k \rangle}{\langle \boldsymbol{u}_i,\ \boldsymbol{u}_i \rangle} \boldsymbol{u}_i \\
&= \boldsymbol{a}_k - \sum_{i=1}^{k-1} \frac{\langle \boldsymbol{u}_i,\ \boldsymbol{a}_k \rangle}{\langle \boldsymbol{u}_i,\ \boldsymbol{u}_i \rangle} \Big(\sum_{j=1}^{i} c_{j,i} \boldsymbol{a}_j \Big) \\
&= \boldsymbol{a}_k - \sum_{i=1}^{k-1} \sum_{j=1}^{i} \frac{\langle \boldsymbol{u}_i,\ \boldsymbol{a}_k \rangle}{\langle \boldsymbol{u}_i,\ \boldsymbol{u}_i \rangle} c_{j,i} \boldsymbol{a}_j \\
&= \boldsymbol{a}_k - \sum_{j=1}^{k-1} \Big(\sum_{i=j}^{k-1} \frac{\langle \boldsymbol{u}_i,\ \boldsymbol{a}_k \rangle}{\langle \boldsymbol{u}_i,\ \boldsymbol{u}_i \rangle} c_{j,i} \Big) \boldsymbol{a}_j
\end{aligned} \tag{3.4}$$

It remains to be shown that the inner products $\langle \boldsymbol{u}_i,\ \boldsymbol{a}_k \rangle$ and $\langle \boldsymbol{u}_i,\ \boldsymbol{u}_i \rangle$ can be expressed in terms of $\mathsf{Cofactor}$ and $\boldsymbol{C}'$ for all $i \in [k-1]$.

$$\begin{aligned}
\langle \boldsymbol{u}_i,\ \boldsymbol{a}_k \rangle &= \langle \sum_{l \in [i]} c_{l,i} \boldsymbol{a}_l,\ \boldsymbol{a}_k \rangle \\
&= \sum_{l \in [i]} c_{l,i} \langle \boldsymbol{a}_l,\ \boldsymbol{a}_k \rangle
\end{aligned} \tag{3.5}$$

$$\begin{aligned}
\langle \boldsymbol{u}_i,\ \boldsymbol{u}_i \rangle &= \langle \boldsymbol{u}_i,\ \sum_{l \in [i]} c_{l,i} \boldsymbol{a}_l \rangle \\
&= \sum_{l \in [i]} c_{l,i} \langle \boldsymbol{u}_i,\ \boldsymbol{a}_l \rangle \\
&= \sum_{l \in [i]} \sum_{p \in [i]} c_{l,i} c_{p,i} \langle \boldsymbol{a}_p,\ \boldsymbol{a}_l \rangle
\end{aligned} \tag{3.6}$$

Finally, we use equation 3.4 to find expressions for $c_{j,k}$ for all $j \in [k-1]$:

$$\begin{aligned}
\boldsymbol{u}_k &= c_{k,k}\boldsymbol{a}_k + \sum_{j=1}^{k-1} \underbrace{\Big(-\sum_{i=j}^{k-1} \frac{\langle \boldsymbol{u}_i,\ \boldsymbol{a}_k \rangle}{\langle \boldsymbol{u}_i,\ \boldsymbol{u}_i \rangle} c_{j,i} \Big)}_{c_{j,k}} \boldsymbol{a}_j \\
&= c_{k,k}\boldsymbol{a}_k + \sum_{j=1}^{k-1} c_{j,k}\boldsymbol{a}_j \\
&= \sum_{j \in [k]} c_{j,k}\boldsymbol{a}_j \\
c_{j,k} &:= -\sum_{i=j}^{k-1} \frac{\langle \boldsymbol{u}_i,\ \boldsymbol{a}_k \rangle}{\langle \boldsymbol{u}_i,\ \boldsymbol{u}_i \rangle} c_{j,i} \qquad (3.7) \\
&= -\sum_{i=j}^{k-1} \frac{\sum_{l \in [i]} c_{l,i} \langle \boldsymbol{a}_l,\ \boldsymbol{a}_k \rangle}{\sum_{l \in [i]} \sum_{p \in [i]} c_{l,i} c_{p,i} \langle \boldsymbol{a}_p,\ \boldsymbol{a}_l \rangle} c_{j,i}
\end{aligned}$$

This concludes the proof by induction, hence equation (3.3) holds for all $k \in [N]$. Moreover, the proof yields the necessary equations to compute $\boldsymbol{C}'$. Next, we show how this relates to $\boldsymbol{Q}$ and in particular to $\boldsymbol{C}$, noting that $\|\boldsymbol{u}_i\| = \sqrt{\langle \boldsymbol{u}_i,\ \boldsymbol{u}_i \rangle}$.

$$\begin{aligned}
\boldsymbol{Q} &= \begin{bmatrix} \frac{\boldsymbol{u}_1}{\|\boldsymbol{u}_1\|} & \frac{\boldsymbol{u}_2}{\|\boldsymbol{u}_2\|} & \cdots & \frac{\boldsymbol{u}_N}{\|\boldsymbol{u}_N\|} \end{bmatrix} \\
&= \boldsymbol{A} \underbrace{\begin{pmatrix} \frac{c_{1,1}}{\|\boldsymbol{u}_1\|} & \frac{c_{1,2}}{\|\boldsymbol{u}_1\|} & \cdots & \frac{c_{1,N}}{\|\boldsymbol{u}_1\|} \\ & \frac{c_{2,2}}{\|\boldsymbol{u}_2\|} & \cdots & \frac{c_{2,N}}{\|\boldsymbol{u}_2\|} \\ & & \ddots & \vdots \\ \boldsymbol{0} & & & \frac{c_{N,N}}{\|\boldsymbol{u}_N\|} \end{pmatrix}}_{\boldsymbol{C}}
\end{aligned}$$

It follows from the definition of $\boldsymbol{R}$ that its entries can be computed efficiently.

$$\begin{aligned}
R_{i,j} &= \langle \boldsymbol{e}_i,\ \boldsymbol{a}_j \rangle \\
&= \langle \frac{\boldsymbol{u}_i}{\|\boldsymbol{u}_i\|},\ \boldsymbol{a}_j \rangle \\
&= \frac{\langle \boldsymbol{u}_i,\ \boldsymbol{a}_j \rangle}{\|\boldsymbol{u}_i\|}
\end{aligned}$$

Finally, we use $\boldsymbol{R}'$ to denote $\boldsymbol{R}$ before dividing by the norm of $\boldsymbol{u}_i$, i.e.:

$$R'_{i,j} = \langle \boldsymbol{u}_i,\ \boldsymbol{a}_j \rangle$$

3.3 From Equations to Algorithm

The derivations from Section 3.2, in particular equations 3.5, 3.6 and 3.7, implicitly give an algorithm to compute the factorised QR decomposition.

The first stage of Algorithm 1 computes $\boldsymbol{C}'$ and $\boldsymbol{R}'$ row-by-row in N iterations.

$$R[i,j] \leftarrow \langle \boldsymbol{u}_i,\ \boldsymbol{a}_j \rangle, \qquad C[i,j] \leftarrow c_{i,j}$$

Before we proceed, we justify storing $\langle \boldsymbol{u}_i,\ \boldsymbol{u}_i \rangle$ in the diagonal of R. More concretely, we use the following result to show that $\sqrt{R[k,k]} = \|\boldsymbol{u}_k\|$:

$$\begin{aligned}
\langle \boldsymbol{u}_i,\ \boldsymbol{u}_i \rangle &= \sum_{l \in [i]} c_{l,i} \langle \boldsymbol{u}_i,\ \boldsymbol{a}_l \rangle \\
&= c_{1,i} \underbrace{\langle \boldsymbol{u}_i,\ \boldsymbol{a}_1 \rangle}_{0} + c_{2,i} \underbrace{\langle \boldsymbol{u}_i,\ \boldsymbol{a}_2 \rangle}_{0} + \cdots + \underbrace{c_{i,i}}_{1} \langle \boldsymbol{u}_i,\ \boldsymbol{a}_i \rangle \\
&= \langle \boldsymbol{u}_i,\ \boldsymbol{a}_i \rangle
\end{aligned}$$

The inner products being 0 is a direct result of the orthogonalisation process. Despite this result, equation 3.6 is used to compute the diagonal of R (line 8) instead of equation 3.5 (which is computationally cheaper). Equation 3.6 corrects for (potential) loss of orthogonality as a result of floating-point errors, resulting in better accuracy in practice.

The last stage of the algorithm (lines 9 and 10) 'normalises' C and R by dividing the kth row of both by $\sqrt{R[k,k]} = \|\boldsymbol{u}_k\|$.

Algorithm 1 Factorised Gram-Schmidt (Naive)

1: **procedure** F-GS(Cofactor: Matrix) ▷ Cofactor contains the inner products
2: $C, R \leftarrow \mathbf{I}_N$, Matrix$[N, N]$
3: **for** $k \in [N]$ **do**
4: **for** $i \in [k-1]$ **do**
5: $R[i,k] \leftarrow \sum_{l \in [i]} C[l,i] \cdot \mathsf{Cofactor}[l,k]$ ▷ Eq 3.5
6: **for** $j \in [k-1]$ **do**
7: $C[j,k] \leftarrow -\sum_{i=j}^{k-1} \frac{R[i,k]}{R[i,i]} C[j,i]$ ▷ Eq 3.7
8: $R[k,k] \leftarrow \sum_{p \in [k]} \sum_{l \in [k]} C[p,k] \cdot C[l,k] \cdot \mathsf{Cofactor}[l,p]$ ▷ Eq 3.6
9: NORMALISE(C, diag(R)) ▷ diag extracts the diagonal of a matrix.
10: NORMALISE(R, diag(R))
11: **return** R, C

12: **function** NORMALISE(M: Matrix, d: Array)
13: **for** $k \in [N]$ **do**
14: **for** $j \leftarrow k$ **until** $N+1$ **do**
15: $M[k,j] \leftarrow \frac{M[k,j]}{\sqrt{d[k]}}$

3.4 Time and Space Complexity

This section analyses the time and space complexities of three different approaches to compute the (factorised) QR decomposition. The first approach reflects on the complexity of **F-GS** and relies on the factorised representation of a join result. In contrast, the second approach considers factorised Gram-Schmidt performed on the listing representation. Finally, we consider the conventional (out-of-database) approach to compute the QR decomposition given the listing representation.

Throughout this section we use N to denote the number of features (after encoding). Clearly we have $N \leq n \cdot |\mathbf{D}|$, where n is the number of variables and $|\mathbf{D}|$ is the database size, i.e. the sum of the number of tuples in the relations of $\mathbf{D}$. More importantly, recall that we use $\mathcal{O}$ to denote the *data complexity*. The data complexity considers n and $|Q|$ (the number of relations in the join query) to be constants, such that $N = \mathcal{O}(|\mathbf{D}|)$. In our case the factors n and $|Q|$ that we leave out are at most quadratic.

3.4.1 Complexity of F-GS

We start by considering the complexity of factorised Gram-Schmidt given Cofactor, noting that the sparse $\mathbf{\Sigma}$ contains the same information as Cofactor.

Lemma 3.3. *Given the $N \times N$ matrix Cofactor for the query Q over database $\mathbf{D}$, **F-GS** needs time $O(N^3)$ to compute the factorised QR decomposition of the matrix defined by $Q(\mathbf{D})$.*

Proof. The factorised QR decomposition consists of computing $\boldsymbol{C}$ and $\boldsymbol{R}$ in N iterations. For each iteration the number of operations is $O(N^2)$, yielding a total runtime of $O(N^3)$. This bound is tight for the computation of $\boldsymbol{C}$, regardless of the potential sparsity of $\mathbf{\Sigma}$. For completeness, we add that the space complexity is proportional to the size of objects C and R and therefore $O(N^2)$. □

In case $Q(\mathbf{D})$ only contains continuous data, then the overall data complexity of **F-GS** is as follows:

Theorem 3.4. *Given a feature extraction join query Q, with continuous features only, over a database $\mathbf{D}$, **F-GS** computes the factorised QR decomposition of $Q(\mathbf{D})$ in time $\mathcal{O}(|\mathbf{D}|^{fhtw(Q)} + |\mathbf{D}| \log |\mathbf{D}|)$.*

Proof. Recall that Proposition 2.2 states that the factorised representation for join result $Q(\mathbf{D})$, containing only continuous data, has size $\mathcal{O}(|\mathbf{D}|^{fhtw(Q)})$. The cofactors can be computed in a single pass over the factorised representation. However, this requires the input data $\mathbf{D}$ to be sorted, which takes log-linear time in the size of $\mathbf{D}$, i.e. $\mathcal{O}(|\mathbf{D}| \log |\mathbf{D}|)$.

Finally, in the continuous case $N = n$, hence by Lemma 3.3 the data complexity of **F-GS** (given the cofactors) is given by $O(n^3) = \mathcal{O}(1)$. □

Finally, in the more general case where $Q(\mathbf{D})$ may contain any number of categorical variables, we have:

Theorem 3.5. *Given a feature extraction join query Q over database* $\mathbf{D}$*,* **F-GS** *computes the factorised QR decomposition of the matrix defined by $Q(\mathbf{D})$ in time $\mathcal{O}(|\mathbf{D}|^{1+\max(fhtw(Q),2)})$.*

Proof. Recall that the cofactors can be computed in a single pass over the factorised representation of $Q(\mathbf{D})$. The sparse encoding of the categorical data in $Q(\mathbf{D})$ may require at most one extra factor linear in the size of the active domains of these categorical variables. More concretely, the factorised representation for heterogeneous $Q(\mathbf{D})$ has size $\mathcal{O}(|\mathbf{D}|^{fhtw(Q)+1})$ [6].

Moreover, it follows from Lemma 3.3 that the additional complexity of **F-GS** is $O(N^3) = \mathcal{O}(|\mathbf{D}|^3)$. This gives an overall complexity of:

$$\mathcal{O}(|\mathbf{D}|^{fhtw(Q)+1} + |\mathbf{D}|^3 + |\mathbf{D}| \log |\mathbf{D}|) = \mathcal{O}(|\mathbf{D}|^{fhtw(Q)+1} + |\mathbf{D}|^3)$$

We can express this more succinctly as $\mathcal{O}(|\mathbf{D}|^{1+\max(fhtw(Q),2)})$. □

3.4.2 Complexity of Listing-Based Approaches

First, we consider factorised Gram-Schmidt on top of the listing representation of the join result. That is in contrast to **F-GS** which relies on a second layer of factorisation to compute the cofactors. Afterwards, we state the complexities of conventional QR decomposition algorithms, such as the classical Gram-Schmidt process. The structure of this section is similar to that of the previous section, however we start by stating the complexity of factorised Gram-Schmidt given the design matrix $\boldsymbol{A}$.

Lemma 3.6. *Given the $m \times N$ matrix $\boldsymbol{A}$, factorised Gram-Schmidt computes the factorised QR decomposition of $\boldsymbol{A}$ in time $O(N^2 \cdot m + N^3)$.*

Proof. Recall that factorised Gram-Schmidt relies on the cofactors of $\boldsymbol{A}$ instead of $\boldsymbol{A}$ itself. The cofactors can be obtained either by computing $O(N^2)$ dot products over the columns of $\boldsymbol{A}$, or by (pre-)computing $\boldsymbol{A}^\top \boldsymbol{A}$, i.e. by (matrix) multiplying two rectangular matrices. Both approaches to compute the cofactors are $O(N^2 \cdot m)$, in addition to the factorised process itself which Lemma 3.3 states is $O(N^3)$. □

Recall that Proposition 2.2 states that the listing representation of the join result $Q(\mathbf{D})$ has size $\mathcal{O}(|\mathbf{D}|^{\rho^*(Q)})$, where ρ^* is the fractional edge cover number. We use this result to characterise factorised Gram-Schmidt on a materialised join result consisting solely of continuous data.

Proposition 3.7. *Given the matrix $\boldsymbol{A} = Q(\mathbf{D})$ for a feature extraction query Q, with continuous features only, over a database* $\mathbf{D}$*, factorised Gram-Schmidt computes the factorised QR decomposition of $\boldsymbol{A}$ in time $\mathcal{O}(|\mathbf{D}|^{\rho^*(Q)})$.*

Proof. The design matrix $\boldsymbol{A}$ defined by $Q(\mathbf{D})$ has dimensions $m \times N$. The dimensions of $\boldsymbol{A}$ have bounds $m = \mathcal{O}(|\mathbf{D}|^{\rho^*(Q)})$ and $N = n = \mathcal{O}(1)$. By combining these bounds with Lemma 3.6, we obtain the data complexity $\mathcal{O}(|\mathbf{D}|^{\rho^*(Q)})$. □

We extend this result to include categorical data, such that:

Proposition 3.8. *Given the join result $Q(\mathbf{D})$ of a query Q over database $\mathbf{D}$, factorised Gram-Schmidt computes the factorised QR decomposition of the matrix $\boldsymbol{A}$ defined by $Q(\mathbf{D})$ in time $\mathcal{O}(|\mathbf{D}|^{\rho^*(Q)+2})$.*

Proof. The dimensions of $\boldsymbol{A}$ have bounds $m = \mathcal{O}(|\mathbf{D}|^{\rho^*(Q)})$ and $N = \mathcal{O}(|\mathbf{D}|)$. However, recall that $\boldsymbol{A}$ must be one-hot encoded, which is possible in time $O(N \cdot m)$. Together with Lemma 3.6, this gives a complexity of $O(N^2 \cdot m + N^3 + N \cdot m) = O(N^2 \cdot m + N^3)$. By substituting the bounds for the dimensions of $\boldsymbol{A}$, we get:

$$\mathcal{O}(|\mathbf{D}|^2 \cdot |\mathbf{D}|^{\rho^*(Q)} + |\mathbf{D}|^3) = \mathcal{O}(|\mathbf{D}|^{\rho^*(Q)+2} + |\mathbf{D}|^3)$$

Since $\rho^*(Q) \geq 1$, this can be simplified to $\mathcal{O}(|\mathbf{D}|^{\rho^*(Q)+2})$. □

Finally, we state the complexity of the conventional methods used to compute the QR decomposition. Under conventional methods we include the Gram-Schmidt process, Householder reflections, and Given's rotations approaches. In contrast to the factorised Gram-Schmidt process, the conventional methods do not make use of the cofactors and instead rely on the materialised $\boldsymbol{A}$.

Proposition 3.9. *Given the join result $Q(\mathbf{D})$ of a query Q over database $\mathbf{D}$, conventional methods compute the QR decomposition of the matrix $\boldsymbol{A}$ defined by $Q(\mathbf{D})$ in time $\mathcal{O}(|\mathbf{D}|^{\rho^*(Q)+2})$.*

Proof. The conventional methods all have computational complexity $O(N^2 \cdot m)$ [17]. The dimensions of $\boldsymbol{A}$ have bounds $m = \mathcal{O}(|\mathbf{D}|^{\rho^*(Q)})$ and $N = \mathcal{O}(|\mathbf{D}|)$. By substituting these bounds, we get: $\mathcal{O}(|\mathbf{D}|^2 \cdot |\mathbf{D}|^{\rho^*(Q)}) = \mathcal{O}(|\mathbf{D}|^{\rho^*(Q)+2})$. □

Recall that for the class of frequently occurring acyclic queries $fhtw(Q) = 1$. Therefore, above results imply that for an acyclic query Q, **F-GS** can compute the QR decomposition of a continuous-only join result $Q(\mathbf{D})$ in time log-linear in the size of the input database $\mathbf{D}$. If we consider categorical data for the same class of queries, **F-GS** needs time cubic in the size of $\mathbf{D}$ — regardless of the size of the join result $Q(\mathbf{D})$ and thus of the size of design matrix $\boldsymbol{A}$! An alternative characterisation is that the computation time (in terms of data complexity) of **F-GS** is quadratic in $|\mathbf{D}|$ plus cubic in the number of features.

This can be arbitrarily better than the state-of-the-art approaches that first materialise $Q(\mathbf{D})$ as a relation. Recall that for acyclic queries, $\rho^*(Q)$ can be as large as $|Q|$, the number of joined relations. Therefore, conventional methods may require time exponential in $|Q|$ for the same class of queries — even in the continuous case. Finally, the data complexity of factorised Gram-Schmidt applied to the materialised join result is identical to the data complexity of conventional (out-of-database) methods. This is not unexpected, because factorised Gram-Schmidt is designed to exploit the second layer of factorisation to compute the cofactors.

3.5 Applications of F-GS

This section describes some applications of the factorised QR decomposition which is computed by **F-GS**. We assume that the factorised QR decomposition of an $m \times N$

matrix $\boldsymbol{A}$ is given, such that:

$$\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{R} = \boldsymbol{A}\boldsymbol{C}\boldsymbol{R}$$

With $N \times N$ matrices $\boldsymbol{C}$ and $\boldsymbol{R}$ materialised, however, with both $\boldsymbol{Q}$ and $\boldsymbol{A}$ symbolically.

3.5.1 Doubly Factorised Linear Least Squares

The linear least squares problem is introduced in Section 2.2. Recall that for the in-database scenario considered, a linear model is given as:

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \tag{3.8}$$

Where $\boldsymbol{A}, \boldsymbol{b} = Q(\mathbf{D})$, i.e. the result of some feature extraction query over database $\mathbf{D}$. In Section 2.3.2 the QR approach to solving equation 3.8 is described.

$$\begin{aligned} \boldsymbol{d} &:= \boldsymbol{Q}^\top \boldsymbol{b} \\ \boldsymbol{R}\hat{\boldsymbol{x}} &= \boldsymbol{d} \end{aligned} \tag{3.9}$$

Backward substitution is used to efficiently solve the system of equation 3.9, however, this requires computing $\boldsymbol{d} = \boldsymbol{Q}^\top \boldsymbol{b}$. Using the ideas of **F-GS**, we can exploit that $\boldsymbol{b}$ is a column in database $\mathbf{D}$ and avoid materialising $\boldsymbol{Q}$ altogether.

$$\begin{aligned} \boldsymbol{d} &= -(\boldsymbol{A}\boldsymbol{C})^\top \boldsymbol{b} \\ &= -\boldsymbol{C}^\top \boldsymbol{A}^\top \boldsymbol{b} \\ &= -\boldsymbol{C}^\top \begin{bmatrix} \langle \boldsymbol{a}_1,\ \boldsymbol{b} \rangle \\ \langle \boldsymbol{a}_2,\ \boldsymbol{b} \rangle \\ \vdots \\ \langle \boldsymbol{a}_N,\ \boldsymbol{b} \rangle \end{bmatrix} \end{aligned}$$

The dot products $\langle \boldsymbol{a}_i,\ \boldsymbol{b} \rangle$ can be computed by FAQs without materialising $\boldsymbol{A}$. In fact, $\boldsymbol{A}^\top \boldsymbol{b}$ should be considered as part of (or an extension of) Cofactor.

Next, we give the pseudocode for **F-GS**$_{LS}$, an extension of **F-GS** which solves linear least squares using the factorised QR decomposition. The algorithm closely follows the method described in Section 2.3.2.

Algorithm 2 Linear Least Squares extension of **F-GS**

procedure F-LLS(Cofactor, Cofactor$_{LS}$) ▷ Cofactor$_{LS}$ contains $\boldsymbol{A}^\top \boldsymbol{b}$
  $R, C \leftarrow$ F-GS(Cofactor)
  $d \leftarrow$ Array$[N]$
  for $k \in [N]$ **do**
    $d[k] \leftarrow \sum_{i \in [k]} C[i,k] \cdot \text{Cofactor}_{LS}[i]$
  return BackwardSubstitute(R, d)

First, the factorised QR decomposition ($\boldsymbol{C}$ and $\boldsymbol{R}$) is obtained by performing factorised Gram-Schmidt. Next, a factorised computation is used to calculate $\boldsymbol{d} = \boldsymbol{Q}^\top \boldsymbol{b}$. Finally, $\boldsymbol{R}\hat{\boldsymbol{x}} = \boldsymbol{d}$ is solved for $\hat{\boldsymbol{x}}$ using backward substitution.

Corollary 3.10 (Theorem 3.5)**.** *Given a join query Q over database $\mathbf{D}$ such that $\boldsymbol{A}, \boldsymbol{b}$ is defined by $Q(\mathbf{D})$, $\mathsf{F\text{-}GS}_{LS}$ can solve the least squares problem $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ in $\mathcal{O}(|\mathbf{D}|^{1+\max(fhtw(Q),2)})$.*

Proof. Computing an entry of $\boldsymbol{d}$ is possible in time linear in N, hence computing $\boldsymbol{d}$ is $O(N^2)$ overall. Similarly, backward substitution is quadratic in the size of the linear system (N). Therefore the data complexity from Theorem 3.5 applies. □

3.5.2 Singular-Value Decomposition

The *singular-value decomposition* (SVD) is an important matrix decomposition.

Definition 3.11. A thin SVD of a matrix $\boldsymbol{A} \in \mathbb{R}^{m \times N}$ is of form $\boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}$, with orthogonal matrices $\boldsymbol{U} \in \mathbb{R}^{m \times N}$, $\boldsymbol{V} \in \mathbb{R}^{N \times N}$, and diagonal matrix $\boldsymbol{\Sigma} \in \mathbb{R}^{N \times N}$.

Despite the identical notation, this matrix $\boldsymbol{\Sigma}$ is unrelated to the Sigma matrix used for cofactors. Applications of the (thin) SVD include matrix approximation, principle component analysis and computing the rank of a matrix.

First, we show that a thin SVD of $\boldsymbol{A}$ can be obtained from the factorised QR decomposition. Next, a more efficient method is proposed which exploits that $\boldsymbol{R}$ is upper triangular. We start by showing a property of orthogonal matrices.

Lemma 3.12. *The matrix product of two orthogonal matrices $\boldsymbol{X} \in \mathbb{R}^{p \times q}$ and $\boldsymbol{Y} \in \mathbb{R}^{q \times r}$ is an orthogonal matrix.*

Proof. In what follows $\boldsymbol{I}$ denotes a rectangular identity matrix (unless $p = r$).

$$\begin{aligned}(\boldsymbol{X}\boldsymbol{Y})^\top(\boldsymbol{X}\boldsymbol{Y}) &= \boldsymbol{Y}^\top\boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{Y} \\ &= \boldsymbol{Y}^\top\boldsymbol{Y} = \boldsymbol{I}\end{aligned}$$

□

Corollary 3.13 (Theorem 3.5)**.** *Given a join query Q over database $\mathbf{D}$ such that $\boldsymbol{A}$ is defined by $Q(\mathbf{D})$, a factorised SVD of $\boldsymbol{A}$ can be obtained in $\mathcal{O}(|\mathbf{D}|^{1+\max(fhtw(Q),2)})$.*

Proof. First, the SVD of $\boldsymbol{R}$ is computed such that $\boldsymbol{R} = \boldsymbol{U}_R\boldsymbol{\Sigma}\boldsymbol{V}^\top$.

$$\begin{aligned}\boldsymbol{A} &= \boldsymbol{A}\boldsymbol{C}\boldsymbol{R} \\ &= \boldsymbol{A}\boldsymbol{C}\boldsymbol{U}_R\boldsymbol{\Sigma}\boldsymbol{V}^\top \\ &= (\boldsymbol{A}\boldsymbol{C}\boldsymbol{U}_R)\boldsymbol{\Sigma}\boldsymbol{V}^\top\end{aligned}$$

Recall that $\boldsymbol{R}$ is $N \times N$ and therefore the SVD of $\boldsymbol{R}$ can be computed in $O(N^3)$ [17]. Moreover, $\boldsymbol{U}_R$ is $N \times N$ such that $\boldsymbol{U} := \boldsymbol{A}\boldsymbol{C}\boldsymbol{U}_R$ is $m \times N$. Finally, using Lemma 3.12 and that $\boldsymbol{Q} = \boldsymbol{A}\boldsymbol{C}$ is orthogonal, we conclude that $\boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$ is a valid SVD. Moreover, the data complexity of $\mathsf{F\text{-}GS}$ (Theorem 3.5) applies. □

Computing the Factorised SVD

A promising method is found in a slight adaptation of an existing procedure used to compute the SVD for matrices. The SVD is typically computed using a two-phase approach, of which the first phase is to bring the matrix into a bidiagonal form. In the second phase the SVD of the bidiagonal matrix is computed. For matrices with $m \gg N$ it is advantageous to use LHC bidiagonalisation which relies on the QR decomposition as an additional first step [17]. By plugging in the factorised QR decomposition we obtain the following procedure:

1. Compute the factorised QR decomposition of $\boldsymbol{A}$:

$$\boldsymbol{A} = \boldsymbol{A}\boldsymbol{C}\boldsymbol{R}$$

2. Compute the Golub-Kahan bidiagonalisation of $\boldsymbol{R}$:

$$\boldsymbol{B} = \boldsymbol{U}_R^\top \boldsymbol{R} \boldsymbol{V}_R \tag{3.10}$$

   Where $\boldsymbol{U}_R$ and $\boldsymbol{V}_R$ are orthogonal and $\boldsymbol{B}$ is bidiagonal.

3. Compute the SVD of $\boldsymbol{B}$:

$$\boldsymbol{B} = \boldsymbol{U}_B \boldsymbol{\Sigma} \boldsymbol{V}_B^\top \tag{3.11}$$

The second step requires $O(N^3)$ floating-point operations (flops) for $N \times N$ matrix $\boldsymbol{R}$. Numerous iterative algorithms can be used for the second phase (i.e. the third step), e.g. a variant of the QR algorithm which requires $O(N^2)$ flops [17].

To show how the SVD can be obtained, we start by using that $\boldsymbol{U}_R \boldsymbol{U}_R^\top = \boldsymbol{I}$ and $\boldsymbol{V}_R \boldsymbol{V}_R^\top = \boldsymbol{I}$ such that:

$$\begin{aligned}
\boldsymbol{R} &= (\boldsymbol{U}_R \boldsymbol{U}_R^\top) \boldsymbol{R} (\boldsymbol{V}_R \boldsymbol{V}_R^\top) \\
&= \boldsymbol{U}_R (\boldsymbol{U}_R^\top \boldsymbol{R} \boldsymbol{V}_R) \boldsymbol{V}_R^\top && \{ \text{ By Eq 3.10 } \} \\
&= \boldsymbol{U}_R \boldsymbol{B} \boldsymbol{V}_R^\top
\end{aligned}$$

Substituting this result in the factorised QR decomposition of $\boldsymbol{A}$ yields:

$$\begin{aligned}
\boldsymbol{A} &= \boldsymbol{A}\boldsymbol{C}\boldsymbol{R} \\
&= \boldsymbol{A}\boldsymbol{C}\boldsymbol{U}_R \boldsymbol{B} \boldsymbol{V}_R^\top && \{ \text{ By Eq 3.11 } \} \\
&= \boldsymbol{A}\boldsymbol{C}\boldsymbol{U}_R (\boldsymbol{U}_B \boldsymbol{\Sigma} \boldsymbol{V}_B^\top) \boldsymbol{V}_R^\top) \\
&= (\boldsymbol{A}\boldsymbol{C}\boldsymbol{U}_R \boldsymbol{U}_B) \boldsymbol{\Sigma} (\boldsymbol{V}_B^\top \boldsymbol{V}_R^\top)
\end{aligned}$$

By Lemma 3.12, the following definitions of $\boldsymbol{U}$ and $\boldsymbol{V}$ constitute an SVD for $\boldsymbol{A}$:

$$\begin{aligned}
\boldsymbol{U} &:= \boldsymbol{A}\boldsymbol{C}\boldsymbol{U}_R \boldsymbol{U}_B \\
\boldsymbol{V}^\top &:= \boldsymbol{V}_B^\top \boldsymbol{V}_R^\top
\end{aligned}$$

Therefore, given the factorised QR decomposition (or rather the cofactors) this approach can compute the factorised SVD in $O(N^3)$, i.e. independent of m.

3.5.3 Cholesky Decomposition

Definition 3.14. The Cholesky decomposition of a symmetric positive-definitive matrix $\boldsymbol{M}$ is a decomposition of the form

$$\boldsymbol{M} = \boldsymbol{L}\boldsymbol{L}^\top$$

Where $\boldsymbol{L}$ is lower triangular and has positive diagonal entries.

There is a known relation between the QR decomposition of $\boldsymbol{A}$ and the Cholesky decomposition of $\boldsymbol{A}^\top\boldsymbol{A}$.

Proposition 3.15. *Given a (factorised) QR decomposition $\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{R}$, a Cholesky decomposition for $\boldsymbol{A}^\top\boldsymbol{A}$ immediately follows.*

Proof.

$$\begin{aligned}\boldsymbol{A}^\top\boldsymbol{A} &= (\boldsymbol{Q}\boldsymbol{R})^\top\boldsymbol{Q}\boldsymbol{R} \\ &= \boldsymbol{R}^\top(\boldsymbol{Q}^\top\boldsymbol{Q})\boldsymbol{R} \\ &= \boldsymbol{R}^\top\boldsymbol{R}\end{aligned}$$

Recall that $R_{k,k} = \sqrt{\|\boldsymbol{u}_k\|} > 0$ and $\boldsymbol{R}$ is upper triangular, therefore, it follows from Definition 3.14 that this is a valid Cholesky decomposition. □

Therefore, the Cholesky decomposition of $\boldsymbol{A}^\top\boldsymbol{A}$ is essentially obtained for free. Even though the Cholesky decomposition is best known for its application to solve linear least squares, it has applications in multiple other problems including non-linear optimisation and Monte Carlo simulation.

3.5.4 Moore-Penrose Inverse

Definition 3.16. The Moore-Penrose inverse of $\boldsymbol{A}$ (denoted $\boldsymbol{A}^\dagger$) is defined for any full-rank $m \times N$ matrix $\boldsymbol{A}$.

$$\boldsymbol{A}^\dagger := (\boldsymbol{A}^\top\boldsymbol{A})^{-1}\boldsymbol{A}^\top$$

The advantage of this generalised inverse is that it can be computed over non-square matrices (i.e. when $\boldsymbol{A}$ is not invertible). The factorised QR decomposition can be used to avoid inverting $(\boldsymbol{A}^\top\boldsymbol{A})^{-1}$.

$$\begin{aligned}\boldsymbol{A}^\dagger &= (\boldsymbol{A}^\top\boldsymbol{A})^{-1}\boldsymbol{A}^\top \\ &= ((\boldsymbol{Q}\boldsymbol{R})^\top\boldsymbol{Q}\boldsymbol{R})^{-1}(\boldsymbol{Q}\boldsymbol{R})^\top \\ &= (\boldsymbol{R}^\top(\boldsymbol{Q}^\top\boldsymbol{Q})\boldsymbol{R})^{-1}\boldsymbol{R}^\top\boldsymbol{Q}^\top \\ &= \boldsymbol{R}^{-1}\boldsymbol{R}^{-\top}\boldsymbol{R}^\top\boldsymbol{Q}^\top \\ &= \boldsymbol{R}^{-1}\boldsymbol{Q}^\top \\ &= \boldsymbol{R}^{-1}(\boldsymbol{C}^\top\boldsymbol{A}^\top)\end{aligned}$$

In practice we may want to postpone (or rather avoid) inverting $\boldsymbol{R}$, and instead rely on algebraic rewrites within the context of the application.

Example. As an example, we show how the (QR) solution to linear least squares can be derived from the Moore-Penrose inverse.

$$\begin{aligned}
\boldsymbol{A}\boldsymbol{x} &= \boldsymbol{b} \\
\hat{\boldsymbol{x}} &= \boldsymbol{A}^{\dagger}\boldsymbol{b} \\
&= (\boldsymbol{R}^{-1}\boldsymbol{Q}^{\top})\boldsymbol{b} \\
\boldsymbol{R}\hat{\boldsymbol{x}} &= \boldsymbol{Q}^{\top}\boldsymbol{b} \\
&= \boldsymbol{C}^{\top}\boldsymbol{A}^{\top}\boldsymbol{b}
\end{aligned}$$

Recall that solving such an upper triangular system is possible in $O(N^2)$ using backward substitution, whereas inverting an upper triangular matrix is $O(N^3)$.

Chapter 4

Implementation

Chapter 3 describes the theoretical results behind **F-GS**, including an algorithm in pseudocode. This chapter presents implementation details of the system, including optimisations and data structures that were used. In particular, it explains how the sparse Sigma matrix is used to avoid the redundancy introduced in Cofactor as a result of one-hot encoding.

Multiple variants of **F-GS** are considered to outline the improvements and implementation details of the final version. Moreover, the parallelisation of **F-GS** and associated challenges are extensively discussed. The chapter concludes with a detailed description of **F-GS**, tying the preceding sections together. Chapter 5 then reports experiments carried out to benchmark and compare **F-GS** to both competitors and its variants.

4.1 Data Structures

4.1.1 Sigma Matrix

Section 2.5.3 describes a succinct representation of $\mathbf{\Sigma}$ in the presence of both continuous and categorical variables. However, the aggregate engine **AC/DC** [8] only uses this for cofactors involving categorical features. In particular, aggregates for a pair of continuous features are generally non-zero and thus do not affect sparsity. Moreover, the remaining aggregates involving at least one categorical feature are listed as 'coordinates', i.e. $(i, j, \phi_{i,j})$. Therefore, **AC/DC** splits the cofactors in a square

$$\begin{pmatrix} m_{1,1} & m_{1,2} & 0 \\ 0 & 0 & m_{2,3} \\ 0 & m_{3,2} & m_{3,3} \end{pmatrix}$$

(a) Sparse Matrix

$$\begin{array}{l} (1, 1, m_{1,1}), \\ (1, 2, m_{1,2}), \\ (2, 3, m_{2,3}), \\ (3, 2, m_{3,2}), \\ (3, 3, m_{3,3}) \end{array}$$

(b) Coordinate list

Figure 4.1: Example of a sparse matrix stored as a coordinate list

matrix for pairs of continuous features (similar to Cofactor) and uses the sparse *Coordinate list* (COO) for the remaining cofactors involving at least one categorical feature. Figure 4.1 shows the COO representation of a 3×3 sparse matrix.

4.1.2 Ordering the Cofactors

One consequence of using the COO format for the categorical cofactors is that the order of iteration is determined by the ordering of this list. However, as the pseudocode in Algorithm 1 indicates, there are two access patterns for the cofactors. Since we prioritise performance over space efficiency, we use a second data structure for the cofactors to reduce the number of redundant cofactors we access.

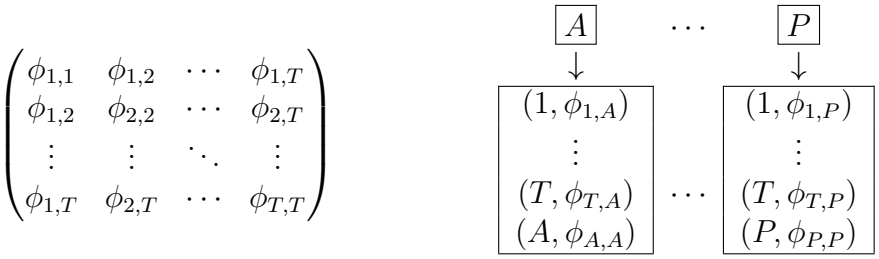
In addition to the COO format, the *list of lists* (LIL) format is used to store the cofactors. Concretely, for each categorical feature a list of pairs (containing the paired feature and aggregate) is created.

We compare the $\mathbf{\Sigma}$ representation described in Section 2.5.3 and the LIL format used in practice to illustrate the differences. Consider an instance with T continuous variables, and one categorical variable with P categories.

$$\mathbf{\Sigma} = \left[\begin{array}{ccc|c} ([], \phi_{1,1}) & \cdots & ([], \phi_{1,T}) & \left\{\begin{array}{c}(A, \phi_{1,A}) \\ \vdots \\ (P, \phi_{1,P})\end{array}\right\} \\ \vdots & \ddots & \vdots & \vdots \\ ([], \phi_{1,T}) & \cdots & ([], \phi_{T,T}) & \left\{\begin{array}{c}(A, \phi_{T,A}) \\ \vdots \\ (P, \phi_{T,P})\end{array}\right\} \\ \hline \left\{\begin{array}{c}(A, \phi_{1,A}) \\ \vdots \\ (P, \phi_{1,P})\end{array}\right\} & \cdots & \left\{\begin{array}{c}(A, \phi_{T,A}) \\ \vdots \\ (P, \phi_{T,P})\end{array}\right\} & \left\{\begin{array}{c}(A, \phi_{A,A}) \\ \vdots \\ (P, \phi_{P,P})\end{array}\right\} \end{array}\right]$$

Figure 4.2: Visualisation of the Sigma matrix for the example instance

Figure 4.2 shows $\mathbf{\Sigma}$ for the same example as Figure 4.3 which is a more accurate visualisation of the cofactors in practice. Figure 4.3 shows that the cofactors are divided into a matrix for the continuous aggregates and the LIL format for categorical aggregates.



(a) Continuous aggregates (b) (Partially) categorical aggregates

Figure 4.3: The cofactors split into two different representations.

Finally, we apply a specific ordering, known as the *colexicographic order* (colex), to the COO format of the cofactors which is optimised for the access pattern used to compute the diagonal of $\boldsymbol{R}$. Within the scope of this dissertation, the following definition of the colex order for pairs $(a, b) \in \mathbb{R}^2$ and $(x, y) \in \mathbb{R}^2$ is sufficient:

$$(a, b) <_{\mathsf{colex}} (x, y) \iff \big(\max(a, b),\, \min(a, b)\big) <_L \big(\max(x, y),\, \min(x, y)\big)$$

Where $<_L$ denotes the lexicographical order (or dictionary order), i.e.:

$$(a, b) <_L (x, y) \iff (a < x) \vee (a = x \wedge b < y)$$

More generally, the colexicographic ordering first obtains a *representative* by sorting the elements in a tuple in non-increasing order. The standard lexicographical ordering is then applied to the representatives.

Tuple	(1,2,3)	(2,4,3)	(1,2,4)	(2,3,3)
Representative	321	432	421	332

This ordering is useful because it corresponds to the nested sums to $k \in \mathbb{N}$:

$$sum \leftarrow \sum_{i \in [k]} \sum_{j \in [k]} m_{i,j}$$

For example, the result of this ordering applied to $\mathbb{N} \times \mathbb{N}$ starts as follows:

$$(1,1),\ (1,2),\ (2,1),\ (2,2),\ (1,3),\ (3,1),\ (2,3),\ (3,2),\ (3,3),\ (1,4),\ (4,1),\ (2,4), \ldots$$

4.1.3 Matrices

Many programming languages and libraries which implement matrices store the underlying matrix as a contiguous array. Our implementation is no different, however, some other details regarding the storage of matrices are relevant for performance.

In particular, a matrix can be stored either row-by-row (row-major) or column-by-column (column-major). For most linear algebra libraries, matrix orientation does not affect the provided interface (e.g. indexing) of the matrix. Nevertheless, the orientation used can affect performance of the algorithm. In particular, matching the orientation to the access pattern (when possible) leads to improved locality and thus better performance for large matrices.

In our implementation all matrices are row-major, except for $\boldsymbol{R}$ which is column-major.

4.2 F-GS Variants

The final implementation **F-GS** contains multiple optimisations and is multithreaded to improve the performance on data matrices with many categorical features. In this section, we describe three earlier (or alternative) variants of **F-GS** which contained fewer optimisations.

For simplicity, the pseudocode in this section assumes that all cofactors are stored in one object. For **Naive F-GS** this is Cofactor, whereas for **Sequential F-GS** and **Triangular F-GS** this is $\mathbf{\Phi}$ which uses the LIL format, i.e. including continuous aggregates. Figure 4.4 shows $\mathbf{\Phi}$ for the working example with T continuous variables and one categorical variable with P categories.

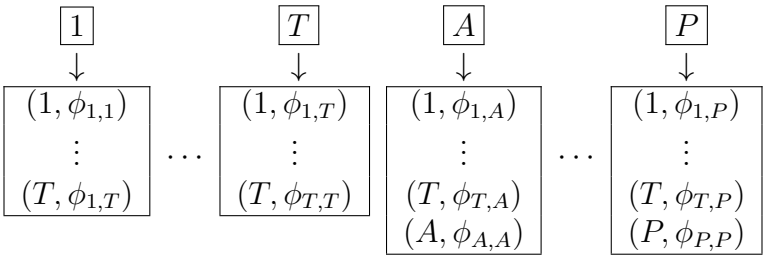


Figure 4.4: Visualisation of $\mathbf{\Phi}$

Naive F-GS

Naive F-GS is a direct (and first) implementation of the rewritten Gram-Schmidt process as described in Algorithm 1. Recall that instead of the sparse encoding of $\mathbf{\Sigma}$, the procedure relied on a square $N \times N$ matrix Cofactor. More concretely, **Naive F-GS** takes the sparse $\mathbf{\Sigma}$ (given by the aggregate engine) as input. It proceeds to create a $N \times N$ matrix Cofactor, in which the aggregates of $\mathbf{\Sigma}$ are placed. This results in a relatively clean and intuitive implementation, however, this matrix is highly redundant and contains many zeroes in contrast to $\mathbf{\Sigma}$. Consequently, calculations which iterate over Cofactor involve a lot of redundant terms, which $\mathbf{\Sigma}$ (or $\mathbf{\Phi}$) avoids.

Sequential F-GS

The single-threaded **Sequential F-GS** is identical to the final system **F-GS** without multithreading. Algorithm 3 shows how the sparse $\mathbf{\Phi}$ is used instead of Cofactor. In particular, the entries of $\boldsymbol{R}$ are computed by iterating over the sparse $\mathbf{\Phi}$ (lines 4 and 7). Finally, we point out that the computation of $\boldsymbol{C}$ (line 6) does not make use of the sparsity of $\mathbf{\Phi}$, and instead iterates over the $N \times N$ matrix $\boldsymbol{R}$.

Algorithm 3 Sequential Factorised Gram-Schmidt

1: **procedure** SEQUENTIAL F-GS($\boldsymbol{\Phi}$: LIL)
2: **for** $k \in [N]$ **do**
3: **for** $i \in [k-1]$ **do**
4: $$R[i,k] \leftarrow \sum_{(l,\phi_{l,k}) \in \boldsymbol{\Phi}[k]}^{l \leq i} C[l,i] \cdot \phi_{l,k}$$
5: **for** $j \in [k-1]$ **do**
6: $$C[j,k] \leftarrow -\sum_{i=j}^{k-1} \frac{R[i,k]}{R[i,i]} \cdot C[j,i]$$
7: $$R[k,k] \leftarrow \sum_{p \in [k]} \sum_{(l,\phi_{l,p}) \in \boldsymbol{\Phi}[p]}^{l \leq k} C[p,k] \cdot C[l,k] \cdot \phi_{l,p}$$

Triangular F-GS

Triangular matrices (and symmetric matrices) are a recurring type of matrix in factorised Gram-Schmidt. Using a traditional matrix to store a triangular or symmetric matrix leads to unnecessary redundancy. Instead, **Triangular F-GS** uses an array for $\boldsymbol{C}$ and $\boldsymbol{R}$, which avoids storing some of the redundant elements. Unlike other sections, this subsection starts indexing at 0 for a much more intuitive description.

Figure 4.5 shows that the elements below the diagonal of an upper triangular matrix are not included in the array. Consequently, indexing becomes more challenging.

$$\begin{pmatrix} m_{0,0} & m_{0,1} & m_{0,2} \\ & m_{1,1} & m_{1,2} \\ \mathbf{0} & & m_{2,2} \end{pmatrix}$$

(a) Upper triangular matrix

0	1	2	3	4	5
$m_{0,0}$	$m_{0,1}$	$m_{0,2}$	$m_{1,1}$	$m_{1,2}$	$m_{2,2}$

(b) Array

Figure 4.5: Example of a 3×3 matrix $\boldsymbol{M}$ in both representations

The functions tri_up and tri_lo are used to convert from matrix index $m_{r,c}$ of an upper and lower triangular matrix (resp.) to the corresponding vector index.

$$\mathsf{tri\_up}(r,c) := N \cdot r - \frac{r(r+1)}{2} + c$$
$$\mathsf{tri\_lo}(r,c) := \frac{r(r+1)}{2} + c$$

It is common (and more efficient) to fill an array (or matrix) with the precomputed vector indices. **Triangular F-GS** uses this method to sparsely store $\boldsymbol{C}$ and $\boldsymbol{R}$.

4.3 Parallelisation

The execution time of **F-GS** is largely spent on computing the entries of $\boldsymbol{C}$ and $\boldsymbol{R}$. Despite the sequential nature of the Gram-Schmidt process, the factorised procedure can benefit from parallelisation. In fact, the entries of $\boldsymbol{C}$ are independent of the other entries in the same row of $\boldsymbol{C}$. Similarly, the entries within the same row of $\boldsymbol{R}$ are independent of each other. Consequently, the computation of entries in a row is conceptually straightforward to parallelise. The two main challenges of parallelising **F-GS** are *synchronisation* and *distribution of work*.

4.3.1 Synchronisation

Each iteration in the computation of $\boldsymbol{C}$ and $\boldsymbol{R}$ consists of three phases:

1. Computing non-diagonal elements in the kth row of $\boldsymbol{R}$.
2. Computing elements in the kth row of $\boldsymbol{C}$.
3. Computing the kth diagonal element of $\boldsymbol{R}$, i.e. $R_{k,k}$.

Broadly speaking, the possible synchronisation points are after (or before) each step. Recall that the entries in $\boldsymbol{C}$ are computed as follows:

$$C[j,k] \leftarrow -\sum_{i=j}^{k-1} \frac{R[i,k]}{R[i,i]} \cdot C[j,i] \tag{4.1}$$

First of all, notice that $C_{1,k}$ depends on all non-diagonal elements in the kth row of $\boldsymbol{R}$. Therefore, the first synchronisation point is after the first step. Similarly, computing the diagonal element $R_{k,k}$ requires the kth row of $\boldsymbol{C}$ to be known. It follows that the next synchronisation point is after the second step. The same synchronisation point is also necessary for computing the $(k+1)$th row of $\boldsymbol{R}$ in the next iteration. Finally, $R_{k,k}$ is first used in the next iteration to compute the $(k+1)$th row of $\boldsymbol{C}$. The first synchronisation point (after step 1) is therefore sufficient, and no third synchronisation point is necessary.

Additionally, computing $R_{k,k}$ takes non-trivial time and the summation is therefore distributed across the threads. Each thread computes a local result which is a 'subtotal' of the value of $R_{k,k}$. The local results are than summed up together to produce $R_{k,k}$. To avoid simultaneous writes to $R_{k,k}$ a *mutex* (lock) is used to provide mutual exclusive access.

To recap, there are two synchronisation points which occur after the first and after the second step. A mutex is used to provide mutual exclusive access to $R_{k,k}$. The synchronisation points are effectively a *barrier* (or meeting point) for the threads; any thread must stop and wait until all threads reach the barrier.

4.3.2 Distribution of Work

One consequence of using barrier synchronisation is that progress is determined by the slowest thread to reach each synchronisation point. To minimise the time threads

$$
\begin{array}{c}
\begin{array}{cccccc}
\text{step 1} & \text{step 2} & \text{step 3} & \text{step 4} & \cdots & \text{step N}
\end{array} \\
\begin{bmatrix}
R_{1,1} & R_{1,2} & R_{1,3} & R_{1,4} & \cdots & R_{1,N} \\
 & R_{2,2} & R_{2,3} & R_{2,4} & \cdots & R_{1,N} \\
 & & R_{3,3} & R_{3,4} & \cdots & R_{3,N} \\
 & & & R_{4,4} & \cdots & R_{4,N} \\
 & & & & \ddots & \vdots \\
 & & & & & R_{N,N}
\end{bmatrix}
\end{array}
$$

Figure 4.6: Distribution of work for $\boldsymbol{R}$ with three threads, visualised by using a different colour (blue, green, red) for each thread. The computational burden of the diagonal entries (in black) is shared over the threads.

spend waiting at the barrier, it is important to balance the distribution of work over the threads. The difficulty is caused by the varying costs of computing entries within the same row in $\boldsymbol{C}$ and $\boldsymbol{R}$. For example, computing $R[1,k] := C[1,1] \cdot \phi_{1,k}$ is much cheaper than computing $R_{k-1,k} := C[1,k-1] \cdot \phi_{1,k} + \cdots + C[k-1,k-1] \cdot \phi_{k-1,k}$.

A common strategy is to partition the work, i.e. the row of k elements, into contiguous blocks such that each partition requires roughly the same amount of computation time. This approach generally has desirable properties, in particular good data locality. However, in our case estimating the costs of each entry is non-trivial and the actual computation time of an entry could depend on many factors including hardware and compiler optimisations. Moreover, the sparsity of $\boldsymbol{\Sigma}$ poses an additional challenge to partition the computational burden evenly.

Instead, we rely on a much simpler approach which does not require estimating the computational cost. Instead of partitioning the rows into contiguous blocks, the threads alternate in computing the next entry, resulting in the interleaved access pattern shown in Figure 4.6. This approach (approximately) balances the workload of computing $\boldsymbol{C}$ and $\boldsymbol{R}$ over the available threads with no added complexity. It follows from equation 4.1 that the entries within a row of $\boldsymbol{C}$ are of increasingly lower computational cost. It can be similarly shown that the entries within a row of $\boldsymbol{R}$ are of increasingly higher computational cost.

However, one disadvantage is that the threads have reduced spatial locality (i.e. for the column-major $\boldsymbol{R}$). Moreover, threads are more likely to (almost) simultaneously access adjacent memory locations which may result in *false sharing* [18]. False sharing occurs when threads access independent variables which are stored in the same cache line; the smallest unit managed by the caching mechanism. For example, when a thread modifies a variable in a particular cache line, other threads that access different variables in the same cache line will be forced to reload the entire cache line.

4.4 Detailed Description of F-GS

Algorithm 4 Detailed Factorised Gram-Schmidt

```
1: procedure F-GS(Cofactor: Matrix, Σ: CoordinateList)
2:     OrderColex(Σ)
3:     Φ := ToListOfLists(Σ)

4:     for tid ∈ [#THREADS] do
5:         CreateThread(DoWork, tid, Cofactor, Σ, Φ)
6:     JoinThreads()

7:     Normalise(C, diag(R))
8:     Normalise(R, diag(R))
9:     return R, C
```

As detailed in Section 4.1, **F-GS** takes as input a $T \times T$ matrix Cofactor containing the aggregates for pairs of continuous variables and a coordinate list representation of $\mathbf{\Sigma}$ with the remaining cofactors. Recall that T denotes the number of continuous features. The procedure starts by applying the colexicographical order to $\mathbf{\Sigma}$ and constructing $\mathbf{\Phi}$ as a copy of $\mathbf{\Sigma}$ in the LIL-format. Next, all the worker threads are created with a unique thread identifier *tid* and the different cofactor structures. The procedure waits for all created threads to finish their work and terminate, before finally normalising C and R (lines 7 and 8).

The DoWork procedure implements the multithreaded version of the computation for $\boldsymbol{C}$ and $\boldsymbol{R}$. The first two stages compute: I) the non-diagonal elements in the kth row of $\boldsymbol{R}$; and II) the elements in the kth row of $\boldsymbol{C}$. In the third stage, each thread computes a subtotal of the diagonal element $R_{k,k}$ in two steps. The first step iterates over the cofactors for pairs of continuous variables stored in the Cofactor. The second step iterates over the colexicographically ordered list $\mathbf{\Sigma}$ to sum up the part of the (partially) categorical cofactors allocated to the thread. Finally, the mutex is obtained to safely add the subtotal of each thread to the (shared) $R[k, k]$.

10: **function** DoWork(tid: Int, Cofactor: Matrix, $\mathbf{\Sigma}$: CoordinateList, $\mathbf{\Phi}$: LIL)
11: $step := \#THREADS$
12: **for** $k \in [N]$ **do**
13: **for** $(i := tid;\ i \leq k;\ i+ = step)$ **do**
14: **if** $k \leq T$ **then**
15: $R[i,k] := \sum_{l \in [i]} C[l,i] \cdot \mathsf{Cofactor}[l,k]$
16: **else**
17: $R[i,k] := \sum_{(l,\phi_{l,k}) \in \mathbf{\Phi}[k]}^{l \leq i} C[l,i] \cdot \phi_{l,k}$
18: Synchronise()
19: **for** $(j := tid;\ j < k; j+ = step)$ **do**
20: $C[j,k] := -\sum_{i=j}^{k-1} \frac{R[i,k]}{R[i,i]} \cdot C[j,i]$
21: Synchronise()
22: $D[k] := \sum_{p=start}^{\min(k,T)} \sum_{l=1}^{\min(k,T)} C[p,k] \cdot C[l,k] \cdot \mathsf{Cofactor}[l,p]$
23: **for** $(i := tid;\ i \leq \mathsf{length}(\mathbf{\Sigma});\ i+ = step)$ **do**
24: $(l,\ p,\ \phi_{l,p}) := \mathbf{\Sigma}[i]$
25: **if** $p > k$ **or** $l > k$ **then**
26: **break loop**
27: $D[k]\ += C[p,k] \cdot C[l,k] \cdot \phi_{l,p}$
28: lock()
29: $R[k,k]\ += D[k]$
30: unlock()

Chapter 5

Experiments

Chapter 4 describes the C++ implementation behind **F-GS**, i.e. the factorised computation of a QR decomposition. This chapter reports the experimental results and findings obtained using this implementation. The performance of **F-GS** is compared to two popular systems **Py** (in Python) and **R** (in R). Moreover, a breakdown of the computation time for Algorithm 4 is provided. The speedup gained by the optimisations in **F-GS** relative to earlier implementations are reported. Finally, the scalability of the parallelisation techniques from Section 4.3 is analysed. Chapter 6 then outlines related work on linear algebra and machine learning performed inside the database system.

5.1 Summary of Findings

This section outlines the findings from experiments performed on several real-world and synthetic datasets, using a quad-core machine with 32GB of main memory.

F-GS is the only system able to compute the QR decomposition over the real-world *Retailer* dataset. Competitors ran out of memory while importing the 3.61 billion values in the join result. This limitation applies even when all attributes are considered to be continuous. In contrast, **F-GS** is able to include thousands of features without crashing or taking an unreasonable amount of time to complete.

The speedup of F-GS over Py consistently matches or beats the compression ratio of the factorised representation to the listing representation. This result applies to all problem instances on which **Py** managed to complete, despite relying on the highly optimised LAPACK procedure for QR decomposition.

F-GS can achieve up to $13\times$ **faster performance than Naive F-GS**. Recall that **Naive F-GS** is a variant which does not exploit the sparsity of the cofactors in $\mathbf{\Sigma}$. The single-threaded variant **Sequential F-GS** is $3.8\times$ faster than **Naive F-GS** for the same data matrix. This improvement is a direct result of the sparsity of $\mathbf{\Sigma}$.

Multithreading yields a speedup of over $3\times$ **on a real-world dataset**. Moreover, **F-GS** performs best when the number of threads is the number of logical cores.

Experiments confirm that, *given the cofactors*, performance of F-GS is determined by the number of features and is independent of the number of rows. Given the cofactors, **F-GS** can compute the QR decomposition of matrices with up to 100 features in less than 10 milliseconds.

The speedup of the linear least squares extensions F-GS$_{LS}$ over Py$_{LS}$ is slightly higher than the speedup of F-GS over Py on the same data matrices. Moreover, experiments show that the accuracy of **F-GS**$_{LS}$ and **Py**$_{LS}$ are comparable; the relative differences in sum of squared residuals are $\leq 10^{-10}$.

5.2 Experimental Setup

This section describes the systems included in the experiments and the environment in which the experiments were performed.

5.2.1 Systems

The following systems are benchmarked: out-of-database competitors **Py** and **R**, all variants of **F-GS**, and the state-of-the-art factorised learner **BGD**.

Basic Linear Algebra Subprograms (BLAS) is a specification of routines that provide standard building blocks for performing basic vector and matrix operations [19]. OpenBLAS is an open-source BLAS implementation, which is highly optimised for modern multi-core computer architectures [20].

LAPACK is a numerical linear algebra package which provides efficient procedures for solving systems of linear equations, eigenvalue problems and obtaining matrix decompositions [21]. LAPACK routines delegate as much work as possible to BLAS calls, in order to benefit from the highly optimised implementation.

Py is a system written in Python and uses the popular SciPy stack (specifically NumPy, pandas and SciPy) [22] to implement a fast out-of-database system with capabilities similar to **F-GS**. The implementation relies on SciPy for QR decomposition; which is simply a wrapper for LAPACK.

Similarly, **R** relies on the base library of statistical language R [23] for QR decomposition, where a flag is used to enable using LAPACK instead of R's default implementation. For perspective, the LAPACK procedure completed more than 20× faster than the default method for the reported experiments.

The systems **Py**$_{LS}$ and **F-GS**$_{LS}$ denote the linear least squares extensions of **Py** and **F-GS** (resp.). Moreover, **R** is only used for linear least squares instances.

F-GS is the optimised and multithreaded implementation of the factorised Gram-Schmidt process, as detailed in Algorithm 4. Recall the three variants of **F-GS**:
(a) **Naive F-GS** is single-threaded and does not exploit the sparsity of $\boldsymbol{\Sigma}$.
(b) **Sequential F-GS** is a single-threaded implementation.
(c) **Triangular F-GS** uses sparse storage for the upper triangular $\boldsymbol{C}$ and $\boldsymbol{R}$.

Finally, we include **BGD** which is an extension of **AC/DC** which uses batch gradient descent (BGD) to find an approximate solution to linear regression problems [6]. This system uses the Barzilai-Borwein step-size adjustment in combination with the Armijo line search condition. **BGD** uses the stopping conditions outlined in the review article [24] with a threshold of 0.0001.

5.2.2 Environment

All experiments were performed on an Intel® Core™ i7-4770 @ 3.40GHz/64bit/32 GB with Linux 3.13.0/g++6.4.0. The Python implementation (**Py**) we compare against uses Python 3.4.3 and SciPy 1.1.0, and the R implementation (**R**) uses R 3.0.2. The underlying libraries used for QR decomposition are LAPACK 3.5.0 with OpenBLAS 0.2.8.

The systems **F-GS** and **BGD** are compiled as an extension of the aggregate engine **AC/DC** [8] (i.e. to compute $\mathbf{\Sigma}$). Consequently, the compiler flags used to compile **AC/DC** also apply to **F-GS** and **BGD**, most notably -$Ofast$ which enables the highest standard optimisation level. For a complete description and analysis, please refer to Section 3.1 of the dissertation by P. Bigourdan [25].

5.3 Tasks

A problem instance is defined as a combination of a *dataset* and a *configuration*. The datasets are given as natural join queries Q over a database $\mathbf{D}$ consisting of relations with some common attributes. A configuration is used to indicate which attributes are to be excluded or included in the data matrix as either continuous or categorical. The features then consist of all continuous variables and all categories of the categorical variables. In short, a problem instance is a complete specification for a data matrix $\boldsymbol{A}$.

Recall that for categorical variables, one category is dropped to retain full-rank, as explained in Section 2.4. Moreover, every problem instance is extended with an intercept feature (constant value) to represent the reference (i.e. dropped) categories. These details are hereafter left implicit.

QR Decomposition

The first task is to compute a (factorised) QR decomposition of a data matrix $\boldsymbol{A}$. For **F-GS** this entails computing the $N \times N$ matrices $\boldsymbol{C}$ and $\boldsymbol{R}$, such that $\boldsymbol{A} = \boldsymbol{QR} = (\boldsymbol{AC})\boldsymbol{R}$. For **Py** the goal is to obtain the QR decomposition using the internal format of LAPACK; $\boldsymbol{H}$ and $\boldsymbol{\tau}$. The $N \times m$ matrix $\boldsymbol{H}$ stores both $\boldsymbol{R}$ and the Householder reflections that generate $\boldsymbol{Q}$. The array $\boldsymbol{\tau}$ stores N additional scaling factors for the Householder reflections. It should be noted that computing $\boldsymbol{H}$ and $\boldsymbol{\tau}$ takes significantly less time than computing $\boldsymbol{Q}$. Finally, the result returned by LAPACK ($\boldsymbol{H}$, $\boldsymbol{\tau}$) is sufficient to solve linear least squares without materialising $\boldsymbol{Q}$, similarly to the result of **F-GS** ($\boldsymbol{C}$, $\boldsymbol{R}$) as described in Section 3.5.1.

Linear Least Squares

The second task is a direct application of the QR decomposition; that is to solve a linear least squares (LLS) instance. In addition to the data matrix defined by the problem instance, one (continuous) variable is specified as the *label* (or dependent variable). The goal is to compute the least squares estimator for the parameters of the linear model in the features of the data matrix. Moreover, both $\mathbf{Py}_{LS}$ and $\boldsymbol{R}$ rely on the QR decomposition to solve LLS.

5.4 Datasets

Experiments are performed on both real-world and synthetic datasets. This section briefly introduces the datasets and the different configurations of each. First of all, the linear models described are not intended to be accurate predictive models. Instead, they are designed to benchmark and analyse **F-GS** under different circumstances.

The Gram-Schmidt process requires the (encoded) data matrix to be full-rank. Therefore, care must be taken to ensure that no linear dependencies occur in the datasets. Including functional determining variables (e.g. keys) results in linear dependence, unless all functionally determined variables are excluded. It follows that, for example, if a primary key is included, all remaining variables of that table must be excluded.

In all cases, the data matrix is obtained by taking the natural join over all relations in the respective dataset. In the following descriptions underlining is used to denote primary keys, and *cursive* to denote the label used in LLS. For both tasks, the label is not considered as a part of the data matrix. A categorical variable 'attr' is occasionally denoted by 'attr(p)' to make explicit the number of categories (p) included as features. Finally, some extra details regarding the datasets are provided in Appendix A.

Retailer

Retailer is a large real-world dataset used to predict customer demand for products based on many external factors [7][8]. The database consists of five relations:

- Inventory(locn, dateid, ksn, *inventoryunits*). Inventory units of each product (ksn) at a location (locn) on a specific date (dateid).
- Census(zip, males, females, medianage, households, etc.). Demographics per zip code.
- Item(ksn, subcategory, category, categorycluster, price). Product information.
- Location(locn, zip, total_area, sell_area, competitor_distance, etc.). Information on each location (i.e. store).
- Weather(locn, dateid, rain, snow, maxtemp, mintemp, etc.). Weather data per location for each date.

In addition to the functional dependencies resulting from the primary keys, the dataset contains two linear dependencies:
1. Columns 'occupiedhouseunits' and 'households' in Census are identical.
2. For each tuple in Census, 'population' equals the sum of 'males' and 'females'.
To preserve full-rank, the columns 'occupiedhouseunits' and 'population' are excluded. Moreover, we have that 'category' determines 'categorycluster' and 'subcategory' determines 'category'. Therefore, at most one of the three attributes is included.

Five different configurations of the retailer dataset are used in the experiments. The first three $\mathtt{v}_0$, $\mathtt{v}_1$ and $\mathtt{v}_2$ use a smaller (partitioned) dataset, intended to accommodate to the limitations of **Py** and **R**. In practice, both systems run out of memory while one-hot encoding the many categorical features in $\mathtt{v}_2$. $\mathtt{v}_0$ includes 31 features, which are exactly all the continuous variables. $\mathtt{v}_1$ extends $\mathtt{v}_0$ with four categorical variables: rain(2), snow(2), thunder(2) and categorycluster(8). $\mathtt{v}_2$ differs from $\mathtt{v}_1$ in that it includes ksn(3431) and excludes categorycluster. $\mathtt{v}_3$ is identical to $\mathtt{v}_2$ but uses the full dataset, resulting in slighly more products, i.e. ksn(3653). Finally, $\mathtt{v}_4$ extends $\mathtt{v}_3$ with dateid(124).

Favorita

Favorita is a public dataset which was released as part of a competition to accurately forecast product sales [26]. The dataset consists of 6 tables:

- Sale(date(1684), store(54), item(4036), *unit_sales*, onpromotion(3))
- Holiday(date, holiday_type(6), locale(3), locale_id(24), transferred(2))
- Item(item, family(33), itemclass(334), perishable(2))
- Oil(date, oilprice)
- Store(store, city(22), state(16), store_type(5), cluster(17))
- Transcation(date, store, transactions)

This dataset is included to benchmark **F-GS** with multiple different categorical variables with a large number of categories. One single configuration is used for the dataset which includes all variables as their true type (i.e. continuous or categorical). Most importantly, this implies that the problem instance contains multiple functional dependencies. Consequently, the data matrix is guaranteed to be rank deficient. Nevertheless, the benchmark results are a useful addition and offer new insights in the performance of **F-GS**.

Housing

Housing is a series of synthetic datasets which represent the textbook example of predicting the housing price market using linear regression [7]. The datasets contain multiple data sources which are relevant for the price of a house. Each relation

		Housing			*Retailer*					*Favorita*
		$\mathtt{v}_0$	$\mathtt{v}_1$	$\mathtt{v}_2$	$\mathtt{v}_0$	$\mathtt{v}_1$	$\mathtt{v}_2$	$\mathtt{v}_3$	$\mathtt{v}_4$	
Variables	Continuous	27	20	16	31	31	31	31	31	3
	Categorical	0	7	11	0	4	4	4	5	14
Features		27	44	110	31	45	3468	3684	3808	6244
Size	Listing	0.68M – 1.47G			774M			3.61G		2.30G
	Factorised	0.68M – 3.57M			37M			169M		377M
Compression		1× – 411×			20.9×			21.4×		6.11×

Table 5.1: Summary of the available configurations.

	Housing										
Scale Factor	1	2	3	4	5	6	7	8	9	10	11
Listing	0.68M	2.70M	24.3M	43.2M	101M	219M	397M	518M	820M	1.01G	1.47G
Factorised	0.68M	1.00M	1.40M	1.68M	1.98M	2.24M	2.57M	2.81M	3.08M	3.32M	3.57M
Compression	1×	2.69×	17.4×	25.7×	51.1×	97.6×	154×	184×	266×	305×	411×

Table 5.2: Sizes and compression of the *Housing* datasets

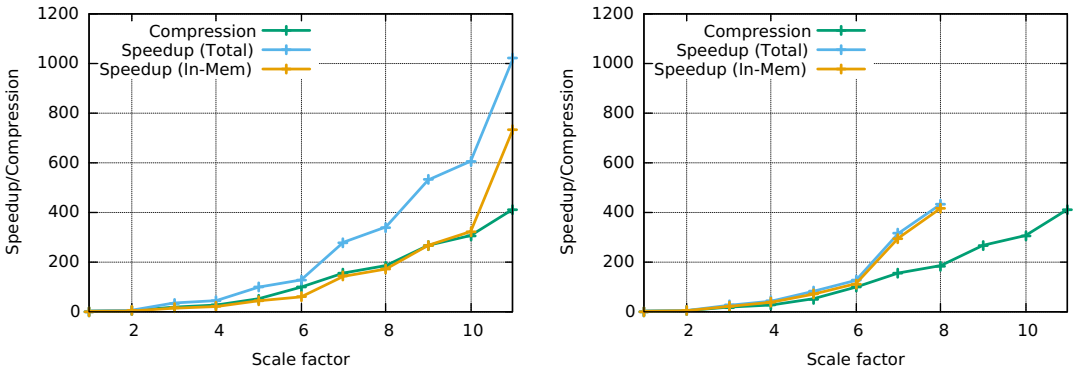
uses the **postcode** attribute as a primary key, which is also used to obtain the join result. *Housing* consists of six relations:

- House(postcode(25000), *price*, livingarea, nbbathrooms, nbbedrooms, etc.)
- Demographics(postcode, averagesalary, crimesperyear, unemployment, etc.)
- Institution(postcode, typeeducation, size)
- Restaurant(postcode, openinghours, pricerange)
- Shop(postcode, pricerange, openinghours, tesco, sainsburys, ms)
- Transport(postcode, nbbuslines, nbtrainstations, distancecitycentre)

There are 25,000 distinct postcodes in every relation. The datasets are created with a scale factor S used to determine the number of tuples per postcode in the relations. Concretely, we generate S tuples in House and Shop, $S/2$ tuples in Restaurant, $\log_2 S$ tuples in Institution, and 1 tuple in Demographics and Transport [7]. The number of records in the join result is therefore given by $\frac{S^3}{2} \log_2 (S) \cdot 25,000$.

Three different versions of the datasets are used with varying number of categorical features. *Housing* $\mathtt{v}_0$ includes all 27 variables as continuous features. $\mathtt{v}_1$ includes six boolean variables (tesco, sainsburys, ms, house, flat and bungalow) and typeeducation(12) as categorical features. Finally, $\mathtt{v}_2$ builds on $\mathtt{v}_1$ by additionally including nbtrainstations(20), nbbuslines(20), nbbedrooms(10) and nbbathrooms(10) as categorical variables — even though they are clearly not.

Table 5.1 summarises the datasets and their respective configurations. Table 5.2 provides details on the sizes and compression ratio of the join result of the *Housing* datasets for scale factors 1 until 11. The compression ratio of a dataset is defined as the size of the factorised representation versus the size of the listing representation of the join result. For example, a compression ratio of 2 means that the *number*



(a) *Housing* $\mathtt{v}_0$ (b) *Housing* $\mathtt{v}_2$

Figure 5.3: Performance of **F-GS** compared to **Py** for two *Housing* configurations.

of values in the factorised representation is half the number of values in the listing representation.

5.5 Experimental Results

This section reports the experimental results obtained for **F-GS** and competing systems. The goal of all experiments, except for the last, is to compute the QR decomposition. The last experiment compares the performance of the linear least squares extensions of all systems. All experiments were performed using 8 threads (i.e. the number of logical cores) for **F-GS** (+ **AC/DC**) and OpenBLAS (used by **Py** and **R**), unless explicitly stated otherwise.

5.5.1 QR Performance

This section compares the performance of **F-GS** to **Py** for computing the QR decomposition. Two different performance metrics are reported: the total execution includes reading the input data and all processing, whereas the in-memory performance excludes reading. The in-memory speedup reflects on the performance benefits of factorised QR decomposition over standard QR decomposition algorithms which are listing representation based.

Figure 5.3 shows the speedup of **F-GS** over **Py** and the compression ratio as a function of the scale factor S. The plots show that both the total and in-memory speedup match or beat the compression ratio. The in-memory speedup is significantly lower than the total speedup for $\mathtt{v}_0$, however the extra categorical features in $\mathtt{v}_2$ close the gap between in-memory and total speedup. One explanation is that the time spent on one-hot encoding and computing the QR decomposition increase significantly, whereas the import time stays the same.

5.5.2 Comparison of F-GS Variants

F-GS was designed with multiple optimisations to improve the performance on matrices with many categorical features. In this section, we analyse the impact of

Speedup	**Naive F-GS**	**Sequential F-GS**	**Triangular F-GS**
Retailer $\mathtt{v}_3$	13.66×	3.58×	1.89×
Retailer $\mathtt{v}_4$	11.41×	3.26×	1.75×
Favorita	6.26×	2.56×	1.27×

Table 5.4: QR decomposition speedup of **F-GS** over its three variants

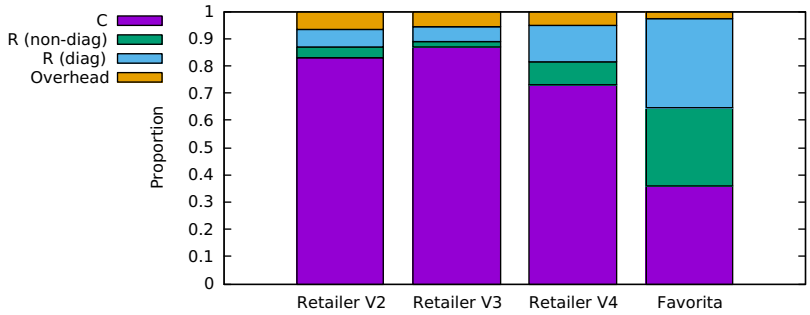


Figure 5.5: Breakdown of the computation time of $\boldsymbol{C}$ and $\boldsymbol{R}$. Overhead represents the fraction of execution time not spent on the computation of $\boldsymbol{C}$ and $\boldsymbol{R}$.

these improvements by comparing **F-GS** to the three variants detailed in Section 4.2. Table 5.4 shows the performance of **F-GS** relative to the variants for three different problem instances. The reported speedups are based on the time spent on computing the QR decomposition, that is given $\boldsymbol{\Sigma}$.

Perhaps the most surprising result is that **F-GS** outperforms **Triangular F-GS** in all experiments. Recall that both **Triangular F-GS** and **F-GS** are multithreaded, and that the single difference is that **Triangular F-GS** uses an array to store only the non-zero halves of triangular matrices $\boldsymbol{C}$ and $\boldsymbol{R}$. Moreover, the performance gap is not related to artefacts of parallelisation. In fact, **Sequential F-GS** is almost twice as fast (1.92×) as **Triangular F-GS** performed with one single thread (on *Retailer* $\mathtt{v}_3$). Therefore, a likely explanation for this performance difference is the indirection caused by the non-linear indices for these sparse triangular matrices.

5.5.3 Breakdown of Factorised Decomposition

The bottleneck of **F-GS** (once $\boldsymbol{\Sigma}$ is computed) is the main loop which computes $\boldsymbol{C}$ and $\boldsymbol{R}$. The main loop can be broken down into three parts; computing $\boldsymbol{C}$, computing non-diagonal elements of $\boldsymbol{R}$, and computing the diagonal of $\boldsymbol{R}$. Synchronisation and execution time outside the main loop are considered to be overhead.

Figure 5.5 shows a breakdown of the computation time of the main loop for four different data matrices. The times are shown as a fraction of the total computation time spent on QR decomposition, i.e. adding up to 1. For the included *Retailer* instances ($\mathtt{v}_2$, $\mathtt{v}_3$ and $\mathtt{v}_4$), the majority of the execution time is spent on calculating $\boldsymbol{C}$. A likely reason is that the computation for $\boldsymbol{R}$ exploits the sparsity of $\boldsymbol{\Sigma}$, whereas the computation for $\boldsymbol{C}$ does not. In particular, the calculation for an entry $R_{i,j}$ iterates over the cofactors in $\boldsymbol{\Sigma}$. In contrast, an entry $C_{i,j}$ is calculated using an iteration over the (dense) matrix $\boldsymbol{R}$. Moreover, *Favorita*, as opposed to *Retailer*,

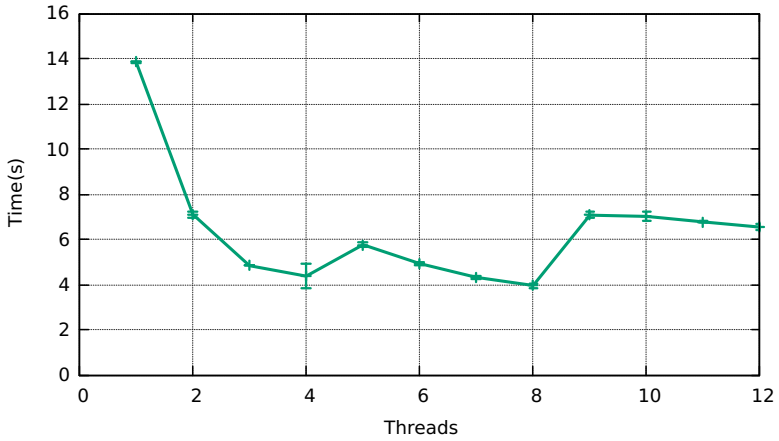


Figure 5.6: QR time of **F-GS** for *Retailer* $\mathtt{v}_2$ as a function of the number of threads.

contains many different categorical variables resulting in a more dense $\mathbf{\Sigma}$. This explains the more even distribution of computation time over the three components.

5.5.4 Impact of Parallelisation

In order to evaluate the performance impact of multithreading, we computed the QR decomposition of the same matrix (*Retailer* $\mathtt{v}_2$) with an increasing number of threads.

Figure 5.6 shows the computation time of the QR decomposition as a function of the number of threads. First of all, the results show that using 2 and 3 threads yields a speedup of respectively 1.95 and 2.86 — which is almost optimal. However, for 4 threads the speedup is 3.02, which clearly indicates a drop in improvement. Since the threads work on the same objects, increased contention likely contributes to this decrease.

Secondly, the performance declines when increasing from 4 to 5 and from 8 to 9 threads. The latter is not unexpected, because at this point the number of threads exceeds the number of logical cores. The performance hit at 5 threads is related to the number of physical cores. This behaviour is examined in Section 5.5.4.1.

In general, the repeated experiments resulted in comparable performance for each repetition. However, when using 4 threads, most runs are roughly 4 seconds; yet some runs take significantly longer. More concretely, out of six repetitions, one execution will almost systematically take up to 7 seconds. The reason for this phenomenon, which was not observed for other numbers of threads, is unclear.

The same approach was used to assess the scalability of **Py** on *Housing* $\mathtt{v}_2$ with scale factor $S = 4$ and *Retailer* $\mathtt{v}_1$. Despite a CPU usage of consistently nearly 100% per (logical) core, there was no noticeable influence on the performance at all.

5.5.4.1 Balancing the Workload Distribution

Recall the discussion of Section 4.3 on the performance penalty of barrier synchronisation caused by an unbalanced workload. The distribution of work can be assessed

Threads	std	Computation time per thread (ms)									
4	9.9	3686	3686	3667	3671	–	–	–	–	–	–
5	1300	5697	3334	5712	3351	3308	–	–	–	–	-
6	1015	2685	4649	2643	4619	4631	4624	–	–	–	–
7	656	4017	4004	3978	4003	2266	4005	4005	–	–	–
8	6.0	3654	3644	3658	3657	3645	3644	3648	3655	–	–
9	28	4064	4051	3987	4037	3993	4046	4067	4042	4040	–
10	23	3406	3375	3358	3427	3356	3401	3389	3379	3403	3423
4*	1111	5769	5783	3860	3843	–	–	–	–	–	–

Table 5.7: Distribution and standard deviation (std) of computation time per thread. Times in red denote (relatively) slow threads, whereas green denotes fast threads.

by measuring the actual computation time per thread, i.e. excluding time idle at synchronisation points. In theory, using 5 threads on a quad-core machine results in one physical core being shared amongst two threads, whereas each of the remaining 3 cores is dedicated to a single thread. This can result in two threads receiving less CPU resources, and the remaining threads spending more time waiting at the barrier.

Table 5.7 shows the computation time for each individual thread. The results confirm the unbalanced distribution of work that occurs when increasing from 4 to 5 threads. Two threads clearly spend more time on computations, while others must wait for these threads. This trend clearly continues for 6 and 7 threads. As expected, when increasing to 8 threads the workload becomes balanced again. Using more than 8 threads results in a relatively balanced workload, likely because of more frequent (and fair) scheduling at the operating system level. Nevertheless, using more threads than logical cores incurs a performance penalty and no obvious benefits.

Finally, Table 5.7 includes a 'slow run' with 4 threads, indicated by an asterisk. Even though this does not represent the majority of executions, it occurs frequently enough to not be considered an outlier. The underlying mechanism causing these unbalanced runs is unknown, one possible explanation is that two threads are executed on the same physical core.

5.5.5 End-to-End Linear Least Squares

In this section we report and compare the end-to-end performance of the LLS extensions of all included systems. The times reported include (if required by the system): materialising and exporting the join result, importing the data matrix, processing the data (e.g. sorting and encoding), and solving LLS.

Table 5.8 shows that **F-GS**$_{LS}$ outperforms both **Py**$_{LS}$ and **R** by a factor which matches the compression ratio. Moreover, whereas **Py** and **R** both crash while importing the full *Retailer* dataset, **F-GS** is able to include more than 3,000 categorical features and complete within two minutes. The time **F-GS** spends on solving (including computing the QR decomposition) is marginal compared to the time spent on aggregate computation for all included tasks. However, it should be noted that **F-GS** adds significantly more time than **BGD** when the number of categorical features grows. Nevertheless, **F-GS** obtains an exact solution whereas **BGD** applies gradient descent to obtain an approximate solution.

Retailer		v_0	v_1	v_2	v_3	v_4
Join Representation	Listing	774M	774M	774M	3.614G	3.614G
(#values)	Factorised	37M	37M	37M	169M	169M
Compression	Fact/List	20.9×	20.9×	20.9×	21.4×	21.4×
Features (continuous + categorical)		32	32+14	32+3437	32+3653	32+3777
Listing representation of the query result						
Join Computation (PSQL)		50.63	50.63	50.63	216.56	216.56
Py$_{LS}$	Export/Import	137.74	137.74	137.74	OOM	OOM
	Process	0.66	5.16	OOM	–	–
	QR	16.97	28.14	–	–	–
	Solve LLS	2.14	9.38	–	–	–
	Total	**208.14**	**231.05**	–	–	–
R	Export/Import	280.72	280.72	280.72	OOM	OOM
	Process	3.41	18.07	OOM	–	–
	QR+Solve	22.71	46.04	–	–	–
	Total	**357.47**	**395.46**	–	–	–
Factorised representation of the query result						
Import + Sort		2.73	2.73	2.73	12.34	12.34
Aggregate Computation		6.78	7.61	65.33	92.98	99.02
BGD	Learn	< 0.01	< 0.01	0.46	0.63	1.4
	Iterations	520	502	459	519	430
F-GS$_{LS}$	QR	< 0.01	< 0.01	3.90	4.80	5.99
	Solve LLS	< 0.01	< 0.01	0.03	0.03	0.03
	Total	**9.51**	**10.34**	**71.99**	**110.15**	**117.38**
Speedup of **F-GS** over	**Py**	21.66×	21.44×	∞	∞	∞
Speedup of **F-GS**$_{LS}$ over	**Py**$_{LS}$	21.89×	22.35×	∞	∞	∞
	R	37.59×	38.25×	∞	∞	∞
	BGD	1×	1×	0.95×	0.96×	0.96×

Table 5.8: Time performance (seconds) comparison for end-to-end LLS using **F-GS**$_{LS}$, **Py**$_{LS}$, **R** and **BGD**. Both **Py**$_{LS}$ and **R** ran out of memory (OOM) while importing (full) *Retailer*.

Chapter 6

Related Work

Previous chapters of this dissertation described the theory and implementation of **F-GS**. This chapter briefly outlines related work on the integration of linear algebra and machine learning into the database management system. Both similarities and differences of earlier approaches with respect to this work are outlined. Finally, Chapter 7 provides a summary of the outcomes and findings of this dissertation, and recommends directions for future research to follow up on these results.

Factorised Learning

This dissertation follows a line of work on factorised learning, e.g. [6], [7], [27]. Factorised learning builds on factorised databases, a new kind of database system pioneered by Olteanu and colleagues [28], and in particular on computing aggregates over the factorised representation [29]. Factorised databases led to several developments; most recently to the idea of learning over normalised databases.

Factorised learning differs from the majority of efforts on scalable machine learning, which instead focuses on designing systems on top of large-scale distributed architectures such as Spark [30] (e.g. MLlib [31]), TensorFlow [32] and SystemML [33][34].

Kumar et al. [27] introduce a new approach that pushes ML computations past key-foreign key joins to avoid materialising the design matrix. Schleich et al. [7] extend factorised learning to building linear regression models over datasets defined by arbitrary join queries. Moreover, their work provides theoretical guarantees, showing that the computational complexity is linear in the size of the factorised join result. Khamis et al. [6] propose a framework for in-database learning, extending the class of factorised learning models to include ridge linear regression, factorization machines and principle component analysis. Moreover, this framework introduces a sparse representation and treatment of categorical variables, which this dissertation relies on.

In-Database Linear Algebra

Performing linear algebra inside the (relational) DBMS is not a new idea, and many different approaches have been proposed. There is an on-going interest in developing a system which supports both linear algebra and relational algebra [35] [36] [37]. The

main challenge is to retain the benefits associated with the standalone counterparts, i.e. the efficiency of OpenBLAS, and capabilities of a DBMS (e.g. indexes, query optimisation) [38].

These approaches recognise the advantages of in-database linear algebra, in particular to avoid the expensive import-export step and to leverage decades of research in relational systems. However, none of these approaches fully exploit the relational structure in the data to the same extent as factorised learning.

Factorised Linear Algebra

Even though little research has been done on factorised linear algebra, some recent research has been done which is conceptually close to the work presented in this dissertation. Chen et al. identify the development overhead of the current trend in factorised learning, which is to manually rewrite specific ML algorithms to push operations past the join [39]. They propose Morpheus, a framework which automatically rewrites linear algebra operations into equivalent relational operations over normalised data. This approach differs from our work in multiple ways. First of all, our long-term goal is to provide a wide range of linear algebra operations which can be used to factorise existing ML algorithms and to develop completely novel algorithms. Morpheus, in contrast, relies on automatically applying a limited set of algebraic rewrite rules, which limits the applications to completely 'supported' algorithms. Moreover, we propose a completely novel factorised linear algebra operation which is not supported by Morpheus. Finally, our work emphasises accommodating categorical variables while avoiding the redundancy of the one-hot encoding.

Chapter 7

Conclusion

This dissertation introduced **F-GS**, a novel approach to compute the QR decomposition of a matrix defined by a join query over normalised relational data. This system uses a two-layer factorisation of the Gram-Schmidt process to avoid materialising the data matrix altogether. The first layer is an algebraic rewrite of the Gram-Schmidt process to expressions in terms of the cofactors. The second layer of our approach is the factorised computation of these cofactors. We have shown that this algorithm has a lower asymptotic complexity than state-of-the-art QR decomposition algorithms, which are based on the highly redundant listing representation as input.

F-GS is implemented in C++ and supports input relations consisting of both continuous and categorical data. Moreover, **F-GS** allows the computation of the QR decomposition to be distributed across the cores of a machine. We have incorporated a sparse encoding for categorical variables in **F-GS**, which resulted in significant performance benefits in practice.

The performance of **F-GS** was investigated and compared against systems based on the industry standard LAPACK, using two real-world datasets and one synthetic dataset. The speedup of **F-GS** against its fastest competitor consistently matched the compression ratio of the factorised representation of the join result compared to the listing representation. Most importantly, **F-GS** was able to compute the QR decomposition of problem instances much larger than any of its competitors. Considering that a real-world retailer dataset we experimented on yields a compression rate of more than 20×, the matching speedup and increased data capacity of **F-GS** have significant practical benefits.

Moreover, **F-GS**$_{LS}$ solves linear least squares on a real-world dataset (partitioned to accommodate limitations of competitors) more than 20× faster than any out-of-database competitor. Finally, we have shown a 13× speedup of the final implementation of **F-GS** over its single-threaded predecessor **Naive F-GS**, which did not exploit the sparse encoding for categorical variables.

This dissertation is the first to introduce a novel fundamental linear algebra operation to factorised databases. We have described multiple applications of the factorised QR decomposition, including solving linear least squares and obtaining the factorised SVD. Whereas linear least squares was fully covered and implemented, for other applications such as the SVD we only scratched the surface. In the final

section of this dissertation, we outline some interesting directions for future research that follow up on the contributions of this dissertation.

7.1 Future Work

Applications of F-GS

This dissertation has only touched upon the many possibilities of the factorised QR decomposition. There are two possible research directions to identify the applications of **F-GS**. The first considers linear algebra operations which can benefit from the factorised QR decomposition. An important example which we already addressed is the factorised singular-value decomposition. Moreover, some applications of the QR decomposition may be relatively unknown because of limited usefulness in general. However, with the clear computational benefits of the factorised QR decomposition these applications could proof incredibly useful in our setting.

The second investigates applications in the more traditional sense, i.e. which problems can be solved. An example we considered is solving linear least squares. However, the factorised QR decomposition and other operations derived from this (including SVD) can potentially be used to solve many other problems in a completely factorised style.

Factorised Linear Algebra

We have already discussed more applications of the factorised QR decomposition. However, there are potentially many useful linear algebra operations which can (on their own) be factorised in a way similar to the factorised Gram-Schmidt process. Therefore, another useful direction for future work is to come up with rewrites of other linear algebra operations completely separate from the QR decomposition.

Implementation Improvements

Even though **F-GS** is multithreaded, the experimental results showed suboptimal results in terms of scalability in the number of threads. The current implementation may suffer from data contention and poor cache performance, for example as a consequence of the interleaved access pattern described in Section 4.3. There is potential for performance improvements by redesigning the algorithm with the modern hardware architecture in mind, see for example Chapter 5 of [18]. For instance, partitioning $\boldsymbol{C}$ and $\boldsymbol{R}$ per thread based on write access could lead to better performance.

Moreover, **F-GS** fails for data matrices which are not full-rank. In fact the Gram-Schmidt process (fundamentally) requires the data matrix to be full-rank. Nevertheless, this is a limitation of **F-GS** which can be addressed by a careful implementation. In particular, it is possible to 'drop' linearly dependent columns on-the-fly during the orthogonalisation process. This can be detected using the norm of the orthogonalised vector ($\boldsymbol{u}_k$) which will be zero (or within some range ϵ to account for floating-point errors).

Finally, **F-GS** expresses the QR computation in terms of the cofactor matrix. The advantage is that these aggregates are used in earlier work on factorised learning and are relatively intuitive. Nevertheless, it is worth looking into the possibility of expressing the subsequent procedure (i.e. computing $\boldsymbol{C}$ and $\boldsymbol{R}$) as FAQs.

Numerical Stability

Numerical stability is an important topic for matrix decompositions. In fact, computationally more expensive decompositions are used over cheaper ones solely because of the improved numerical stability they provide. Moreover, the numerical properties of a decomposition depend on the algorithm used to compute it.

The **F-GS**$_{LS}$ solution to linear least squares was comparable to the result obtained using LAPACK (relative difference in residual sum of squares $\leq 1 \cdot 10^{-10}$) on multiple problem instances. Nevertheless, a rigorous numerical analysis (e.g. of the condition number) of **F-GS** is valuable. Moreover, the implementation can potentially be refined to improve its numerical properties.

Other QR algorithms

Another research direction, related to numerical stability, is rewriting other decomposition algorithms to compute the factorised QR decomposition. Two prominent alternatives to the Gram-Schmidt process are Householder reflections and Given's rotations. Both alternatives are known to have better numerical properties than the Gram-Schmidt process.

Within our research we already attempted to come up with a factorised rewrite of the Householder approach. This resulted in a factorisation of the QR computation itself, however, we did not manage to apply the factorised result without needing to materialise the data matrix. The obtained derivations are included as an appendix for future reference. Moreover, Appendix B outlines the challenges of applying factorised Householder QR to solve linear least squares without materialising the design matrix. Nevertheless, different rewrites may be possible, and exploring alternative algorithms could result in factorised QR decompositions with better (numerical) properties.

Bibliography

[1] Carlos A. Gomez-Uribe and Neil Hunt. The Netflix recommender system: Algorithms, business value, and innovation. *ACM Trans. Management Inf. Syst.*, 6(4):13:1–13:19, 2016.

[2] IBM. Big data solutions. `https://www.ibm.com/it-infrastructure/solutions/big-data`. [Online; last accessed on 2018-08-21].

[3] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design and implementation of the LogicBlox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1371–1382, 2015.

[4] Feng Yan, Olatunji Ruwase, Yuxiong He, and Trishul M. Chilimbi. Performance modeling and scalability optimization of distributed deep learning systems. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Sydney, NSW, Australia, August 10-13, 2015*, pages 1355–1364, 2015.

[5] Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. Towards a unified architecture for in-RDBMS analytics. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 325–336, 2012.

[6] Mahmoud Abo Khamis, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. In-database learning with sparse tensors. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*, pages 325–340, 2018.

[7] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. Learning linear regression models over factorized joins. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 3–18, 2016.

[8] Mahmoud Abo Khamis, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. AC/DC: in-database learning thunderstruck. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning, DEEM@SIGMOD 2018, Houston, TX, USA, June 15, 2018*, pages 8:1–8:10, 2018.

[9] Ahmet Kara and Dan Olteanu. Covers of query results. In *21st International Conference on Database Theory, ICDT 2018, March 26-29, 2018, Vienna, Austria*, pages 16:1–16:22, 2018.

[10] Kevin P. Murphy. *Machine learning - a probabilistic perspective*. Adaptive computation and machine learning series. MIT Press, 2012.

[11] Dan Olteanu and Maximilian Schleich. Factorized databases. *SIGMOD Record*, 45(2):5–16, 2016.

[12] Åke Björck. Solving linear least squares problems by gram-schmidt orthogonalization. *BIT Numerical Mathematics*, 7(1):1–21, 1967.

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

[14] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA*, pages 739–748, 2008.

[15] Dan Olteanu and Jakub Závodný. Size bounds for factorised representations of query results. *ACM Trans. Database Syst.*, 40(1):2:1–2:44, 2015.

[16] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*, pages 37–48. ACM, 2012.

[17] Lloyd N. Trefethen and David Bau. *Numerical linear algebra*. SIAM, 1997.

[18] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel computer architecture - a hardware / software approach*. Morgan Kaufmann, 1999.

[19] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[20] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. AUGEM: automatically generate high performance dense linear algebra kernels on x86 CPUs. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13*, pages 1–12. IEEE, 2013.

[21] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[22] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001. `http://www.scipy.org`, Last accessed on 2018-08-05.

[23] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2015.

[24] Tom Goldstein, Christoph Studer, and Richard G. Baraniuk. A field guide to forward-backward splitting with a FASTA implementation. *CoRR*, abs/1411.3406, 2014.

[25] Pierre-Yves Bigourdan. Distributed and multi-threaded learning of regression models. Master's thesis, University of Oxford, 2016.

[26] Corporación Favorita. Corporación Favorita grocery sales forecasting. `https://www.kaggle.com/c/favorita-grocery-sales-forecasting`, 2018. [Online; last accessed on 2018-08-15].

[27] Arun Kumar, Jeffrey F. Naughton, and Jignesh M. Patel. Learning generalized linear models over normalized data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1969–1984. ACM, 2015.

[28] Dan Olteanu and Jakub Zavodny. Factorised representations of query results: size bounds and readability. In Alin Deutsch, editor, *15th International Conference on Database Theory, ICDT '12, Berlin, Germany, March 26-29, 2012*, pages 285–298. ACM, 2012.

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

[30] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*. USENIX Association, 2010.

[31] Xiangrui Meng, Joseph K. Bradley, Burak Yavuz, Evan R. Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D. B. Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. MLlib: Machine learning in Apache Spark. *Journal of Machine Learning Research*, 17:34:1–34:7, 2016.

[32] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 265–283. USENIX Association, 2016.

[33] Amol Ghoting, Rajasekar Krishnamurthy, Edwin P. D. Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. SystemML: Declarative machine learning on MapReduce.

In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 231–242. IEEE Computer Society, 2011.

[34] Matthias Boehm, Michael Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick Reiss, Prithviraj Sen, Arvind Surve, and Shirish Tatikonda. SystemML: Declarative machine learning on Spark. *PVLDB*, 9(13):1425–1436, 2016.

[35] Andreas Kunft, Alexander Alexandrov, Asterios Katsifodimos, and Volker Markl. Bridging the gap: towards optimization across linear and relational algebra. In *BeyondMR@SIGMOD*, page 1. ACM, 2016.

[36] Dylan Hutchison, Bill Howe, and Dan Suciu. LaraDB: A minimalist kernel for linear and relational algebra computation. In *BeyondMR@SIGMOD*, pages 2:1–2:10. ACM, 2017.

[37] Robert Brijder, Floris Geerts, Jan Van den Bussche, and Timmy Weerwag. On the expressive power of query languages for matrices. In *ICDT*, volume 98 of *LIPIcs*, pages 10:1–10:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

[38] Christopher R. Aberger, Andrew Lamb, Kunle Olukotun, and Christopher Ré. LevelHeaded: Making worst-case optimal joins work in the common case. *CoRR*, abs/1708.07859, 2017.

[39] Lingjiao Chen, Arun Kumar, Jeffrey F. Naughton, and Jignesh M. Patel. Towards linear algebra over normalized data. *PVLDB*, 10(11):1214–1225, 2017.

[40] Robert A. van de Geijn. Notes on Householder QR factorization. `http://www.cs.utexas.edu/users/flame/Notes/NotesOnHouseholderQR.pdf`, 2014. [Online; last accessed on 2018-08-24].

Appendix A

Datasets

Figures A.1, A.2, A.3 show the variable orderings used to compute the factorised representations of the datasets *Retailer*, *Housing*, and *Favorita* (respectively).

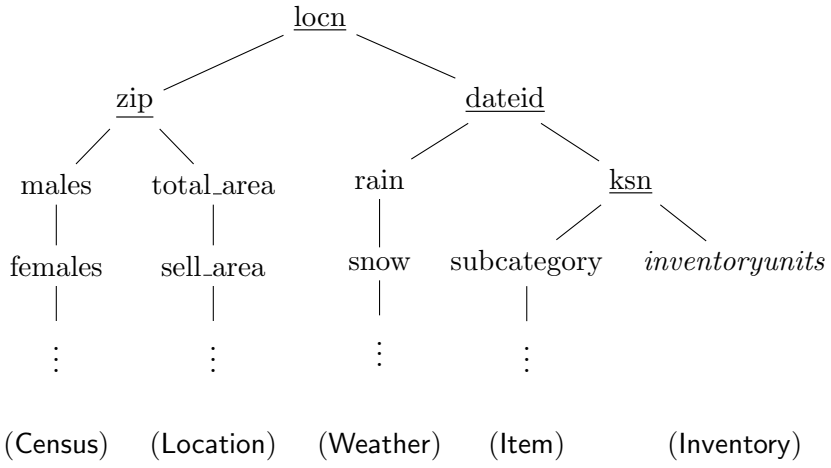


Figure A.1: Visualisation of the variable order for the *Retailer* dataset

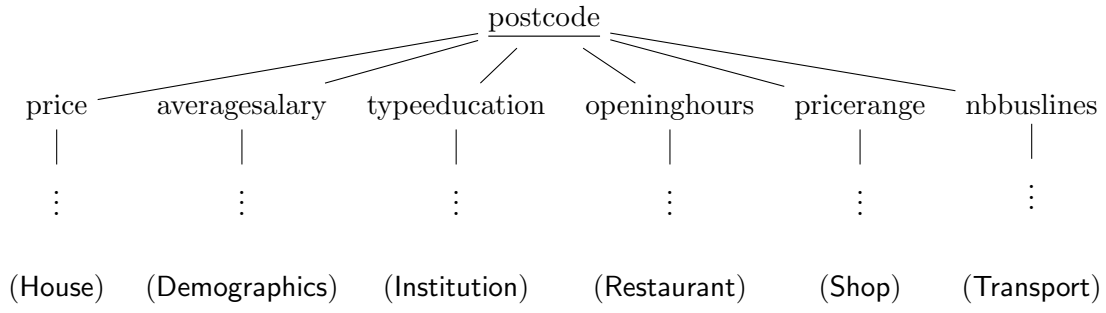


Figure A.2: Visualisation of the variable order for the *Housing* dataset

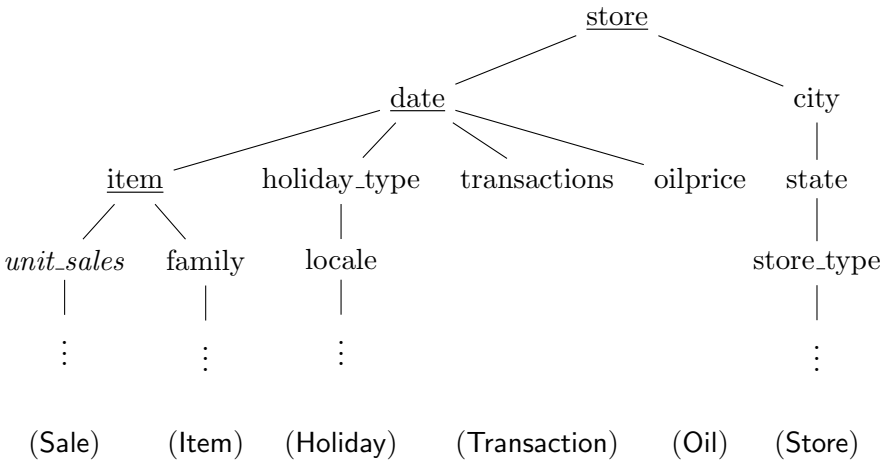


Figure A.3: Visualisation of the variable order for the *Favorita* dataset

Appendix B

Factorised Householder

In this appendix we briefly discuss and present the obtained results (or rather progress) related to the Householder reflections method used to obtain the QR decomposition. In particular, we present an algebraic rewrite of the 'Householder QR' method, which shares similarities with the rewrite of the Gram-Schmidt process described in Chapter 3.

It should be noted that this appendix is included for future reference, and is of lesser quality than earlier chapters of this dissertation. Finally, [40] does a good job at explaining the (out-of-database) Householder QR decomposition.

B.1 Rewriting Householder QR

B.1.1 Outline and definitions

We assume the familiar in-database setting where the matrix $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ is defined by a join query Q over database $\mathbf{D}$. We introduce some notation and definitions to express the rewritten (factorised) expressions.

First of all, we introduce slice notation: $\boldsymbol{A}[i\colon | \, j\colon]$ is the result of dropping the first $i-1$ rows and $j-1$ columns of $\boldsymbol{A}$. We have the following definitions:

$$
\begin{aligned}
\boldsymbol{e}_1 &:= \begin{bmatrix} 1 & 0 & \cdots & 0 \end{bmatrix}^\top \\
\boldsymbol{B}^{(k)} &:= \boldsymbol{Q}_{k-1}\boldsymbol{Q}_{k-2}\ldots\boldsymbol{Q}_1\boldsymbol{A} \\
\boldsymbol{u}_k &:= \boldsymbol{B}^{(k)}[k\colon | \, k] - \left\|\boldsymbol{B}^{(k)}[k\colon | \, k]\right\|\boldsymbol{e}_1 \\
\boldsymbol{Q}_k &= \begin{bmatrix} \boldsymbol{I}_{k-1} & \boldsymbol{0} \\ \boldsymbol{0} & \tilde{\boldsymbol{Q}}_k \end{bmatrix} \\
\tilde{\boldsymbol{Q}}_k &:= \boldsymbol{I} - \frac{2\boldsymbol{u}_k\boldsymbol{u}_k^\top}{\langle \boldsymbol{u}_k,\ \boldsymbol{u}_k \rangle} \\
\boldsymbol{A}[k:j] &= \begin{bmatrix} a_{k,j} & a_{k+1,j} & \ldots & a_{m,j} \end{bmatrix}^\top
\end{aligned}
$$

Note that $\tilde{\boldsymbol{Q}}_k$ is symmetric, from which it follows that $\boldsymbol{Q}$ is symmetric.

We show how some definitions relate to each other:

$$\boldsymbol{B}^{(k)} = \begin{bmatrix} r_{11} & r_{12} & \dots & \dots & \dots & \dots & r_{1n} \\ 0 & r_{22} & \dots & \dots & \dots & \dots & r_{2n} \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & r_{k-1,k-1} & \dots & \dots & r_{(k-1),n} \\ \mathbf{0} & \dots & \dots & \mathbf{0} & \boldsymbol{B}^{(k)}[k: \,|\, k] & \dots & \boldsymbol{B}^{(k)}[k: \,|\, n] \end{bmatrix}$$

$$\boldsymbol{B}^{(k)}[(k-1) : |j] = \begin{bmatrix} r_{k-1,j} & \boldsymbol{B}^{(k)}[k : |j] \end{bmatrix}^\top$$

$$\begin{aligned} \boldsymbol{R} = \boldsymbol{B}^{(n+1)} &= \boldsymbol{Q}_n \boldsymbol{Q}_{n-1} \cdots \boldsymbol{Q}_1 \boldsymbol{A} \\ &= \begin{bmatrix} \boldsymbol{I}_{n-1} & \mathbf{0} \\ \mathbf{0} & \tilde{\boldsymbol{Q}}_n \end{bmatrix} \begin{bmatrix} \boldsymbol{I}_{n-2} & \mathbf{0} \\ \mathbf{0} & \tilde{\boldsymbol{Q}}_{n-1} \end{bmatrix} \cdots \tilde{\boldsymbol{Q}}_1 \boldsymbol{A} \\ &= \begin{bmatrix} \boldsymbol{B}^{(2)}[1,1] & \boldsymbol{B}^{(2)}[1,2] & \boldsymbol{B}^{(2)}[1,3] & \cdots & \boldsymbol{B}^{(2)}[1,n] \\ 0 & \boldsymbol{B}^{(3)}[2,2] & \boldsymbol{B}^{(3)}[2,3] & \cdots & \boldsymbol{B}^{(3)}[2,n] \\ 0 & 0 & \boldsymbol{B}^{(4)}[3,3] & \cdots & \boldsymbol{B}^{(4)}[3,n] \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \boldsymbol{B}^{(n+1)}[n,n] \end{bmatrix} \end{aligned}$$

We use the following property that follows from the definition of the matrix product. For any (compatible) matrices $\boldsymbol{X}$ and $\boldsymbol{Y} = (\boldsymbol{y}_1, \dots, \boldsymbol{y}_p)$, we have:

$$(\boldsymbol{XY})[i : j \,|\, u : v] = \boldsymbol{X}[i : j]\boldsymbol{Y}[: \,|\, u : v] \qquad \text{i.e.:} \quad (\boldsymbol{XY})[i : \,|\, k] = \boldsymbol{X}[i :]\boldsymbol{y}_k$$

We use this to show an essential expression for $\boldsymbol{B}^{(k+1)}[k \,|\, j]$; i.e. the following recursive property:

$$\begin{aligned} \boldsymbol{B}^{(k+1)}[k : \,|\, j] &= \boldsymbol{Q}_k[k :]\boldsymbol{B}^{(k)}[: \,|\, j] \\ &= \begin{bmatrix} \boldsymbol{I}_{k-1} & \mathbf{0}^\top \\ \mathbf{0} & \tilde{\boldsymbol{Q}}_k \end{bmatrix} [k :] \cdot \boldsymbol{B}^{(k)}[: \,|\, j] \\ &= \begin{bmatrix} \mathbf{0} & \tilde{\boldsymbol{Q}}_k \end{bmatrix} \cdot \boldsymbol{B}^{(k)}[: \,|\, j] \\ &= \tilde{\boldsymbol{Q}}_k \boldsymbol{B}^{(k)}[k : \,|\, j] \end{aligned}$$

For completeness, note that here $\mathbf{0}$ has $m - k + 1$ rows and $k - 1$ columns.
Similarly to cofactors for **F-GS**, we introduce the notion of *base terms*, which are expressions that can be efficiently computed using factorised computations.

$$\begin{aligned} \langle \boldsymbol{A}[k: \,|\, i],\ \boldsymbol{A}[k: \,|\, j] \rangle &= \sum_{t=k}^{m} a_{t,i} \cdot a_{t,j} && \forall i, j \in [n]\ \forall k \in [n+1] \\ \langle \boldsymbol{A}[k: \,|\, j],\ \boldsymbol{e}_1 \rangle &= \boldsymbol{A}[k, j] = a_{k,j} && \forall j \in [n]\ \forall k \in [n+1] \end{aligned}$$

Our goal is to rewrite the Householder QR decomposition algorithm to expressions in base terms. More concretely, we set out to (inductively) show that we can express:

$$\boldsymbol{B}^{(k)}[k: \,|\, j] = \boldsymbol{A}[k: \,|\, j] + \sum_{i=1}^{k-1} \alpha_{i,j}^{(k)} \boldsymbol{A}[k: \,|\, i] \qquad \forall k \in [n+1], k-1 \le j \le n$$

Where $\alpha_{i,j}^{(k)}$ denote scalars expressed (entirely) in base terms. Hence, $\boldsymbol{B}^{(k)}[k\colon | j]$ is a linear combination in the (trimmed) columns of $\boldsymbol{A}$.
As an introduction to the idea and the (slightly convoluted) notation we start with finding expressions for $\boldsymbol{B}^{(2)}[2\colon | 2]$ and $\boldsymbol{B}^{(3)}[3\colon | 2]$ as an introduction, before moving to the general case.

B.1.2 A Simplified Example

We consider $\boldsymbol{B}^{(2)}[2\colon | 2]$ and $\boldsymbol{B}^{(3)}[3\colon | 2]$ (instead of the more general $\boldsymbol{B}^{(2)}[2\colon | j]$ and $\boldsymbol{B}^{(3)}[3\colon | j]$) to further simplify the expressions. We start by rewriting the following terms:

$$\begin{aligned}\boldsymbol{u}_1 &= \boldsymbol{a}_1 - \|\boldsymbol{a}_1\|\boldsymbol{e}_1\\ \tilde{\boldsymbol{Q}}_1 &= \boldsymbol{I} - 2\frac{\boldsymbol{u}_1\boldsymbol{u}_1^\top}{\langle \boldsymbol{u}_1,\ \boldsymbol{u}_1\rangle}\end{aligned}$$

Using that $\|\boldsymbol{a}_1\| = \sqrt{\langle \boldsymbol{a}_1,\ \boldsymbol{a}_1\rangle}$ is a base term, we can focus on rewriting $\langle \boldsymbol{u}_1,\ \boldsymbol{u}_1\rangle$:

$$\begin{aligned}\langle \boldsymbol{u}_1,\ \boldsymbol{u}_1\rangle &= \langle \boldsymbol{a}_1 - \|\boldsymbol{a}_1\|\boldsymbol{e}_1,\ \boldsymbol{a}_1 - \|\boldsymbol{a}_1\|\boldsymbol{e}_1\rangle\\ &= \langle \boldsymbol{a}_1,\ \boldsymbol{a}_1\rangle - 2\|\boldsymbol{a}_1\|\langle \boldsymbol{a}_1,\ \boldsymbol{e}_1\rangle + \|\boldsymbol{a}_1\|^2\langle \boldsymbol{e}_1,\ \boldsymbol{e}_1\rangle\\ &= 2\langle \boldsymbol{a}_1,\ \boldsymbol{a}_1\rangle - 2\|\boldsymbol{a}_1\|a_{1,1}\\ c_1 &:= \frac{1}{\langle \boldsymbol{a}_1,\ \boldsymbol{a}_1\rangle - \|\boldsymbol{a}_1\|a_{1,1}} = \frac{2}{\langle \boldsymbol{u}_1,\ \boldsymbol{u}_1\rangle}\end{aligned}$$

Using c_1 we can denote $\boldsymbol{B}^{(2)}[1\colon | 2] = \boldsymbol{B}^{(2)}[\,\colon | 2]$ as:

$$\begin{aligned}\boldsymbol{B}^{(2)}[\,\colon | 2] &= \tilde{\boldsymbol{Q}}_1\boldsymbol{B}^{(1)}[\,\colon | 2] = \tilde{\boldsymbol{Q}}_1\boldsymbol{a}_2\\ &= (\boldsymbol{I} - c_1\boldsymbol{u}_1\boldsymbol{u}_1^\top)\boldsymbol{a}_2\\ &= \boldsymbol{a}_2 - c_1\langle \boldsymbol{u}_1,\ \boldsymbol{a}_2\rangle\boldsymbol{u}_1\\ \langle \boldsymbol{u}_1,\ \boldsymbol{a}_2\rangle &= \langle \boldsymbol{a}_1 - \|\boldsymbol{a}_1\|\boldsymbol{e}_1,\ \boldsymbol{a}_2\rangle\\ &= \langle \boldsymbol{a}_1,\ \boldsymbol{a}_2\rangle - \|\boldsymbol{a}_1\|\langle \boldsymbol{e}_1,\ \boldsymbol{a}_2\rangle\\ &= \langle \boldsymbol{a}_1,\ \boldsymbol{a}_2\rangle - \|\boldsymbol{a}_1\|a_{1,2}\end{aligned}$$

We can now concisely express $\boldsymbol{B}^{(2)}[\,\colon | 2]$ in base terms.

$$\begin{aligned}\boldsymbol{B}^{(2)}[\,\colon | 2] &= \boldsymbol{a}_2 - c_1\big(\langle \boldsymbol{a}_1,\ \boldsymbol{a}_2\rangle - \|\boldsymbol{a}_1\|a_{1,2}\big)\boldsymbol{u}_1\\ &= \boldsymbol{a}_2 - \frac{\langle \boldsymbol{a}_1,\ \boldsymbol{a}_2\rangle - \|\boldsymbol{a}_1\|a_{1,2}}{\langle \boldsymbol{a}_1,\ \boldsymbol{a}_1\rangle - \|\boldsymbol{a}_1\|a_{1,1}}\big(\boldsymbol{a}_1 - \|\boldsymbol{a}_1\|\boldsymbol{e}_1\big)\\ &= \boldsymbol{a}_2 + \alpha_{1,2}^{(2)}\boldsymbol{a}_1 + \beta^{(2)}\boldsymbol{e}_1\\ \alpha_{1,2}^{(2)} &= -\frac{\langle \boldsymbol{a}_1,\ \boldsymbol{a}_2\rangle - \|\boldsymbol{a}_1\|a_{1,2}}{\langle \boldsymbol{a}_1,\ \boldsymbol{a}_1\rangle - \|\boldsymbol{a}_1\|a_{1,1}}\\ \beta^{(2)} &= -\alpha_{1,2}^{(2)}\|\boldsymbol{a}_1\|\end{aligned}$$

Similarly, we find that $\boldsymbol{B}^{(2)}[2\colon | 2]$ can than be expressed as:

$$\begin{aligned}\boldsymbol{B}^{(2)}[2\colon | 2] &= \boldsymbol{a}_2[2\,\colon] + \alpha_{1,2}^{(2)}\boldsymbol{a}_1[2\,\colon] + \beta^{(2)}\boldsymbol{e}_1[2\,\colon]\\ &= \boldsymbol{A}[2\colon | 2] + \alpha_{1,2}^{(2)}A[2\colon | 1]\end{aligned}$$

Hence, $\boldsymbol{B}^{(2)}[2{:}\,|\,2]$ is given by a linear combination of the first and second columns of $\boldsymbol{A}$ after dropping the first row. For illustrative purposes, we show how we could have directly derived this result.

$$\begin{aligned}
\boldsymbol{B}^{(2)}[2{:}\,|\,j] &= (\boldsymbol{Q}_1\boldsymbol{B}^{(1)})[2{:}\,|\,j] = (\boldsymbol{Q}_1\boldsymbol{A})[2{:}\,|\,j] \\
&= (\boldsymbol{I} - c_1\boldsymbol{u}_1\boldsymbol{u}_1^\top)[2:] \cdot \boldsymbol{a}_j \\
&= \big(\boldsymbol{I}[2:] - c_1\boldsymbol{u}_1[2:]\boldsymbol{u}_1^\top\big)\boldsymbol{a}_j \\
&= \boldsymbol{a}_j[2:] - c_1\langle \boldsymbol{u}_1,\ \boldsymbol{a}_1\rangle \boldsymbol{u}_1[2:] \\
&= \boldsymbol{A}[2{:}\,|\,j] - c_1\langle \boldsymbol{u}_1,\ \boldsymbol{a}_1\rangle\big(\boldsymbol{A}[2{:}\,|\,1] - \|\boldsymbol{a}_1\|\boldsymbol{e}_1[2:]\big)
\end{aligned}$$

Next, we do the same for $\boldsymbol{B}^{(3)}[3{:}\,|\,2]$ using the expression for $\boldsymbol{B}^{(2)}[2{:}\,|\,2]$.

$$\begin{aligned}
\boldsymbol{u}_2 &= \boldsymbol{B}^{(2)}[2{:}\,|\,2] - \big\|\boldsymbol{B}^{(2)}[2{:}\,|\,2]\big\|\boldsymbol{e}_1 \\
\langle \boldsymbol{B}^{(2)}[2{:}\,|\,2],\ \boldsymbol{B}^{(2)}[2{:}\,|\,2]\rangle &= \langle \boldsymbol{A}[2{:}\,|\,2] + \alpha_{1,2}^{(2)}\boldsymbol{A}[2{:}\,|\,1],\ \boldsymbol{A}[2{:}\,|\,2] + \alpha_{1,2}^{(2)}\boldsymbol{A}[2{:}\,|\,1]\rangle \\
&= \langle \boldsymbol{A}[2{:}\,|\,2],\ \boldsymbol{A}[2{:}\,|\,2]\rangle + 2\alpha_{1,2}^{(2)}\langle \boldsymbol{A}[2{:}\,|\,2],\ \boldsymbol{A}[2{:}\,|\,1]\rangle + \\
&\quad (\alpha_{1,2}^{(2)})^2\langle \boldsymbol{A}[2{:}\,|\,1],\ \boldsymbol{A}[2{:}\,|\,1]\rangle
\end{aligned}$$

So, we can compute $\big\|\boldsymbol{B}^{(2)}[2{:}\,|\,2]\big\|$ and proceed with $\langle \boldsymbol{u}_2,\ \boldsymbol{u}_2\rangle$ in order to obtain $\tilde{\boldsymbol{Q}}_2$:

$$\begin{aligned}
\langle \boldsymbol{u}_2,\ \boldsymbol{u}_2\rangle &= \langle \boldsymbol{B}^{(2)}[2{:}\,|\,2] - \big\|\boldsymbol{B}^{(2)}[2{:}\,|\,2]\big\|\boldsymbol{e}_1,\ \boldsymbol{B}^{(2)}[2{:}\,|\,2] - \big\|\boldsymbol{B}^{(2)}[2{:}\,|\,2]\big\|\boldsymbol{e}_1\rangle \\
&= \langle \boldsymbol{B}^{(2)}[2{:}\,|\,2],\ \boldsymbol{B}^{(2)}[2{:}\,|\,2]\rangle - 2\big\|\boldsymbol{B}^{(2)}[2{:}\,|\,2]\big\|\langle \boldsymbol{B}^{(2)}[2{:}\,|\,2],\ \boldsymbol{e}_1\rangle \\
&\quad + \big\|\boldsymbol{B}^{(2)}[2{:}\,|\,2]\big\|^2\langle \boldsymbol{e}_1,\ \boldsymbol{e}_1\rangle \\
&= 2\langle \boldsymbol{B}^{(2)}[2{:}\,|\,2],\ \boldsymbol{B}^{(2)}[2{:}\,|\,2]\rangle - 2\big\|\boldsymbol{B}^{(2)}[2{:}\,|\,2]\big\|\langle \boldsymbol{B}^{(2)}[2{:}\,|\,2],\ \boldsymbol{e}_1\rangle \\
\langle \boldsymbol{B}^{(2)}[2{:}\,|\,2],\ \boldsymbol{e}_1\rangle &= \langle \boldsymbol{A}[2{:}\,|\,2] + \alpha_{1,2}^{(2)}A[2{:}\,|\,1],\ \boldsymbol{e}_1\rangle \\
&= \langle \boldsymbol{A}[2{:}\,|\,2],\ \boldsymbol{e}_1\rangle + \alpha_{1,2}^{(2)}\langle \boldsymbol{A}[2{:}\,|\,1],\ \boldsymbol{e}_1\rangle \\
&= a_{2,2} + \alpha_{1,2}^{(2)}a_{2,1} \\
c_2 &:= \frac{2}{\langle \boldsymbol{u}_2,\ \boldsymbol{u}_2\rangle} \\
&= \frac{1}{\langle \boldsymbol{B}^{(2)}[2{:}\,|\,2],\ \boldsymbol{B}^{(2)}[2{:}\,|\,2]\rangle - \big\|\boldsymbol{B}^{(2)}[2{:}\,|\,2]\big\|\langle \boldsymbol{B}^{(2)}[2{:}\,|\,2],\ \boldsymbol{e}_1\rangle}
\end{aligned}$$

We use c_2 to start rewriting $\boldsymbol{B}[2{:}\,|\,2]$ to base terms:

$$\begin{aligned}
\boldsymbol{B}^{(3)}[2{:}\,|\,2] &= \tilde{\boldsymbol{Q}}_2\boldsymbol{B}^{(2)}[2{:}\,|\,2] \\
&= (\boldsymbol{I} - c_2\boldsymbol{u}_2\boldsymbol{u}_2^\top)\boldsymbol{B}^{(2)}[2{:}\,|\,2] \\
&= \boldsymbol{B}^{(2)}[2{:}\,|\,2] - c_2\langle \boldsymbol{u}_2,\ \boldsymbol{B}^{(2)}[2{:}\,|\,2]\rangle\boldsymbol{u}_2 \\
\langle \boldsymbol{u}_2,\ \boldsymbol{B}^{(2)}[2{:}\,|\,2]\rangle &= \langle \boldsymbol{B}^{(2)}[2{:}\,|\,2] - \big\|\boldsymbol{B}^{(2)}[2{:}\,|\,2]\big\|\boldsymbol{e}_1,\ \boldsymbol{B}^{(2)}[2{:}\,|\,2]\rangle \\
&= \langle \boldsymbol{B}^{(2)}[2{:}\,|\,2],\ \boldsymbol{B}^{(2)}[2{:}\,|\,2]\rangle - \big\|\boldsymbol{B}^{(2)}[2{:}\,|\,2]\big\|\langle \boldsymbol{e}_1,\ \boldsymbol{B}^{(2)}[2{:}\,|\,2]\rangle
\end{aligned}$$

Note that there are no new terms in $\langle \boldsymbol{u}_2,\ \boldsymbol{B}^{(2)}[2{:}\,|\,2]\rangle$, hence let:

$$c_2' := c_2\langle \boldsymbol{u}_2,\ \boldsymbol{B}^{(2)}[2{:}\,|\,2]\rangle$$

This allows us to cleanly express $\boldsymbol{B}^{(3)}[2{:}\,|\,2]$ in base terms:

$$\begin{aligned}
\boldsymbol{B}^{(3)}[2{:}\,|\,2] &= \boldsymbol{B}^{(2)}[2{:}\,|\,2] - c_2'\boldsymbol{u}_2 \\
&= \boldsymbol{B}^{(2)}[2{:}\,|\,2] - c_2'\big(\boldsymbol{B}^{(2)}[2{:}\,|\,2] - \big\|\boldsymbol{B}^{(2)}[2{:}\,|\,2]\big\|\boldsymbol{e}_1\big) \\
&= \boldsymbol{A}[2{:}\,|\,2] + \alpha_{1,2}^{(2)}\boldsymbol{A}[2{:}\,|\,1] - c_2'\big(\boldsymbol{A}[2{:}\,|\,2] + \alpha_{1,2}^{(2)}\boldsymbol{A}[2{:}\,|\,1]\big) + c_2'\big\|\boldsymbol{B}^{(2)}[2{:}\,|\,2]\big\|\boldsymbol{e}_1 \\
&= \boldsymbol{A}[2{:}\,|\,2] - c_2'\boldsymbol{A}[2{:}\,|\,2] + \big(\alpha_{1,2}^{(2)} - c_2'\alpha_{1,2}^{(2)}\big)\boldsymbol{A}[2{:}\,|\,1] + c_2'\big\|\boldsymbol{B}^{(2)}[2{:}\,|\,2]\big\|\boldsymbol{e}_1 \\
&= \boldsymbol{A}[2{:}\,|\,2] + \alpha_{2,2}^{(3)}\boldsymbol{A}[2{:}\,|\,2] + \alpha_{1,2}^{(3)}\boldsymbol{A}[2{:}\,|\,1] + \beta^{(3)}\boldsymbol{e}_1 \\
\alpha_{1,2}^{(3)} &= \alpha_{1,2}^{(2)} - c_2'\alpha_{1,2}^{(2)} \\
\alpha_{2,2}^{(3)} &= -c_2' \\
\beta^{(3)} &= -\alpha_{2,2}^{(3)}\big\|\boldsymbol{B}^{(2)}[2{:}\,|\,2]\big\|
\end{aligned}$$

Finally, we find that $\boldsymbol{B}^{(3)}[3{:}\,|\,2]$ can than be expressed as:

$$\begin{aligned}
\boldsymbol{B}^{(3)}[3{:}\,|\,2] &= \big(\boldsymbol{B}^{(3)}[2{:}\,|\,2]\big)[2:] \\
&= \big(\boldsymbol{A}[2{:}\,|\,2] + \alpha_{2,2}^{(3)}\boldsymbol{A}[2{:}\,|\,2] + \alpha_{1,2}^{(3)}\boldsymbol{A}[2{:}\,|\,1] + \beta^{(3)}\boldsymbol{e}_1\big)[2:] \\
&= \boldsymbol{A}[3{:}\,|\,2] + \alpha_{2,2}^{(3)}\boldsymbol{A}[3{:}\,|\,2] + \alpha_{1,2}^{(3)}\boldsymbol{A}[3{:}\,|\,1] + \beta^{(3)}\boldsymbol{e}_1[2:] \\
&= \boldsymbol{A}[3{:}\,|\,2] + \alpha_{2,2}^{(3)}\boldsymbol{A}[3{:}\,|\,2] + \alpha_{1,2}^{(3)}\boldsymbol{A}[3{:}\,|\,1]
\end{aligned}$$

B.1.3 Generalised Expression

We proceed to generalise this, using proof by induction, that is we assume that for some k:

$$\boldsymbol{B}^{(k)}[k{:}\,|\,j] = \boldsymbol{A}[k{:}\,|\,j] + \sum_{i=1}^{k-1}\alpha_{i,j}^{(k)}\boldsymbol{A}[k{:}\,|\,i]$$

We start by rewriting $\boldsymbol{B}^{(k+1)}[k{:}\,|\,j]$ to find the necessary terms.

$$\begin{aligned}
\boldsymbol{u}_k &= \boldsymbol{B}^{(k)}[k{:}\,|\,k] - \big\|\boldsymbol{B}^{(k)}[k{:}\,|\,k]\big\|\boldsymbol{e}_1 \\
\tilde{\boldsymbol{Q}}_k &= \boldsymbol{I} - 2\frac{\boldsymbol{u}_k\boldsymbol{u}_k^\top}{\langle\boldsymbol{u}_k,\ \boldsymbol{u}_k\rangle} \\
\boldsymbol{B}^{(k+1)}[k{:}\,|\,j] &= \tilde{\boldsymbol{Q}}_k\boldsymbol{B}^{(k)}[k{:}\,|\,j] \\
&= \Big(\boldsymbol{I} - 2\frac{\boldsymbol{u}_k\boldsymbol{u}_k^\top}{\langle\boldsymbol{u}_k,\ \boldsymbol{u}_k\rangle}\Big)\boldsymbol{B}^{(k)}[k{:}\,|\,j] \\
&= \boldsymbol{B}^{(k)}[k{:}\,|\,j] - 2\frac{\langle\boldsymbol{u}_k,\ \boldsymbol{B}^{(k)}[k{:}\,|\,j]\rangle}{\langle\boldsymbol{u}_k,\ \boldsymbol{u}_k\rangle}\boldsymbol{u}_k
\end{aligned}$$

We can further expand the terms $\langle \boldsymbol{u}_k,\ \boldsymbol{u}_k \rangle$ and $\langle \boldsymbol{u}_k,\ \boldsymbol{B}^{(k)}[k\colon \mid j] \rangle$

$$
\begin{aligned}
\langle \boldsymbol{u}_k,\ \boldsymbol{B}^{(k)}[k\colon \mid j] \rangle &= \langle \boldsymbol{B}^{(k)}[k\colon \mid k] - \big\|\boldsymbol{B}^{(k)}[k\colon \mid k]\big\| \boldsymbol{e}_1,\ \boldsymbol{B}^{(k)}[k\colon \mid j] \rangle \\
&= \langle \boldsymbol{B}^{(k)}[k\colon \mid k],\ \boldsymbol{B}^{(k)}[k\colon \mid j] \rangle - \big\|\boldsymbol{B}^{(k)}[k\colon \mid k]\big\| \langle \boldsymbol{e}_1,\ \boldsymbol{B}^{(k)}[k\colon \mid j] \rangle \\
&= \langle \boldsymbol{B}^{(k)}[k\colon \mid k],\ \boldsymbol{B}^{(k)}[k\colon \mid j] \rangle - \big\|\boldsymbol{B}^{(k)}[k\colon \mid k]\big\| \boldsymbol{B}^{(k)}[k, j] \\
\langle \boldsymbol{u}_k,\ \boldsymbol{u}_k \rangle &= \langle \boldsymbol{B}^{(k)}[k\colon \mid k] - \big\|\boldsymbol{B}^{(k)}[k\colon \mid k]\big\| \boldsymbol{e}_1,\ \boldsymbol{B}^{(k)}[k\colon \mid k] - \big\|\boldsymbol{B}^{(k)}[k\colon \mid k]\big\| \boldsymbol{e}_1 \rangle \\
&= \langle \boldsymbol{B}^{(k)}[k\colon \mid k],\ \boldsymbol{B}^{(k)}[k\colon \mid k] \rangle - 2\big\|\boldsymbol{B}^{(k)}[k\colon \mid k]\big\| \langle \boldsymbol{e}_1,\ \boldsymbol{B}^{(k)}[k\colon \mid k] \rangle \\
&\quad + \big\|\boldsymbol{B}^{(k)}[k\colon \mid k]\big\|^2 \langle \boldsymbol{e}_1,\ \boldsymbol{e}_1 \rangle \\
&= \langle \boldsymbol{B}^{(k)}[k\colon \mid k],\ \boldsymbol{B}^{(k)}[k\colon \mid k] \rangle - 2\big\|\boldsymbol{B}^{(k)}[k\colon \mid k]\big\| \boldsymbol{B}^{(k)}[k, k] \\
&\quad + \langle \boldsymbol{B}^{(k)}[k\colon \mid k],\ \boldsymbol{B}^{(k)}[k\colon \mid k] \rangle \\
&= 2\langle \boldsymbol{B}^{(k)}[k\colon \mid k],\ \boldsymbol{B}^{(k)}[k\colon \mid k] \rangle - 2\big\|\boldsymbol{B}^{(k)}[k\colon \mid k]\big\| \boldsymbol{B}^{(k)}[k, k]
\end{aligned}
$$

Note how we only need to rewrite two expressions to base terms for a complete rewrite. First, we introduce a more convenient expression for $\boldsymbol{B}^{(k)}[k\colon \mid k]$ using $\alpha^{(k)}_{k,k} = 1$, i.e.:

$$
\begin{aligned}
\boldsymbol{B}^{(k)}[k\colon \mid k] &= \boldsymbol{A}[k\colon \mid k] + \sum_{i=1}^{k-1} \alpha^{(k)}_{i,k} \boldsymbol{A}[k\colon \mid i] = \sum_{i=1}^{k} \alpha^{(k)}_{i,k} \boldsymbol{A}[k\colon \mid i] \\
\langle \boldsymbol{B}^{(k)}[k\colon \mid k],\ \boldsymbol{B}^{(k)}[k\colon \mid j] \rangle &= \langle \sum_{i=1}^{k} \alpha^{(k)}_{i,k} \boldsymbol{A}[k\colon \mid i],\ \boldsymbol{A}[k\colon \mid j] + \sum_{i=1}^{k-1} \alpha^{(k)}_{i,j} \boldsymbol{A}[k\colon \mid i] \rangle \\
&= \sum_{i=1}^{k} \Big(\alpha^{(k)}_{i,k} \langle \boldsymbol{A}[k\colon \mid i],\ \boldsymbol{A}[k\colon \mid j] + \sum_{t=1}^{k-1} \alpha^{(k)}_{t,j} \boldsymbol{A}[k\colon \mid t] \rangle \Big) \\
&= \sum_{i=1}^{k} \Big(\alpha^{(k)}_{i,k} \langle \boldsymbol{A}[k\colon \mid i],\ \boldsymbol{A}[k\colon \mid j] \rangle + \sum_{t=1}^{k-1} \alpha^{(k)}_{i,k} \alpha^{(k)}_{t,j} \langle \boldsymbol{A}[k\colon \mid i],\ \boldsymbol{A}[k\colon \mid t] \rangle \Big)
\end{aligned}
$$

$$
\langle \boldsymbol{e}_1,\ \boldsymbol{B}^{(k)}[k\colon \mid j] \rangle = \boldsymbol{B}^{(k)}[k, j] = \boldsymbol{A}[k, j] + \sum_{i=1}^{k-1} \alpha^{(k)}_{i,j} \boldsymbol{A}[k, i]
$$

Using these general derivations we can succinctly denote:

$$
\begin{aligned}
\langle \boldsymbol{B}^{(k)}[k\colon \mid k],\ \boldsymbol{B}^{(k)}[k\colon \mid k] \rangle &= \sum_{i=1}^{k} \sum_{j=1}^{k} \alpha^{(k)}_{i,k} \alpha^{(k)}_{j,k} \langle \boldsymbol{A}[k\colon \mid i,\ \boldsymbol{A}[k\colon \mid j] \rangle \\
\boldsymbol{B}^{(k)}[k, k] &= \sum_{i=1}^{k} \alpha^{(k)}_{i,k} \boldsymbol{A}[k, i] \\
\langle \boldsymbol{u}_k,\ \boldsymbol{u}_k \rangle &= 2\langle \boldsymbol{B}^{(k)}[k\colon \mid k],\ \boldsymbol{B}^{(k)}[k\colon \mid k] \rangle - 2\big\|\boldsymbol{B}^{(k)}[k\colon \mid k]\big\| \boldsymbol{B}^{(k)}[k, k] \\
&= 2\Big(\sum_{i=1}^{k} \sum_{j=1}^{k} \alpha^{(k)}_{i,k} \alpha^{(k)}_{j,k} \langle \boldsymbol{A}[k\colon \mid i,\ \boldsymbol{A}[k\colon \mid j] \rangle \Big) \\
&\quad - 2\big\|\boldsymbol{B}^{(k)}[k\colon \mid k]\big\| \sum_{i=1}^{k} \alpha^{(k)}_{i,k} \boldsymbol{A}[k, i]
\end{aligned}
$$

Hence, letting $c_{k,j} := -2\frac{\langle \boldsymbol{u}_k,\, \boldsymbol{B}^{(k)}[k\colon \mid j]\rangle}{\langle \boldsymbol{u}_k,\, \boldsymbol{u}_k\rangle}$ we can express $\boldsymbol{B}^{(k+1)}[k\colon \mid j]$ as follows:

$$
\begin{aligned}
c_{k,j} &:= -2\frac{\langle \boldsymbol{u}_k,\, \boldsymbol{B}^{(k)}[k\colon \mid j]\rangle}{\langle \boldsymbol{u}_k,\, \boldsymbol{u}_k\rangle} \\
&= -\frac{\langle \boldsymbol{B}^{(k)}[k\colon \mid k,\, \boldsymbol{B}^{(k)}[k\colon \mid j]\rangle - \big\|\boldsymbol{B}^{(k)}[k\colon \mid k]\big\|\boldsymbol{B}^{(k)}[k,j]}{\langle \boldsymbol{B}^{(k)}[k\colon \mid k,\, \boldsymbol{B}^{(k)}[k\colon \mid k]\rangle - \big\|\boldsymbol{B}^{(k)}[k\colon \mid k]\big\|\boldsymbol{B}^{(k)}[k,k]} \\
\boldsymbol{B}^{(k+1)}[k\colon \mid j] &= \boldsymbol{B}^{(k)}[k\colon \mid j] + c_{k,j}\boldsymbol{u}_k \\
&= \boldsymbol{A}[k\colon \mid j] + \sum_{i=1}^{k-1}\alpha_{i,j}^{(k)}\boldsymbol{A}[k\colon \mid i] + c_{k,j}\big(\boldsymbol{B}^{(k)}[k\colon \mid k] - \big\|\boldsymbol{B}^{(k)}[k\colon \mid k]\big\|\boldsymbol{e}_1\big) \\
&= \boldsymbol{A}[k\colon \mid j] + \sum_{i=1}^{k-1}\alpha_{i,j}^{(k)}\boldsymbol{A}[k\colon \mid i] + c_{k,j}\Big(\sum_{i=1}^{k}\alpha_{i,k}^{(k)}\boldsymbol{A}[k\colon\mid i]\Big) - c_{k,j}\big\|\boldsymbol{B}^{(k)}[k\colon\mid k]\big\|\boldsymbol{e}_1 \\
&= \boldsymbol{A}[k\colon \mid j] + c_{k,j}\alpha_{k,k}^{(k)}\boldsymbol{A}[k\colon \mid k] + \sum_{i=1}^{k-1}\big(\alpha_{i,j}^{(k)} + c_{k,j}\alpha_{i,k}^{(k)}\big)\boldsymbol{A}[k\colon \mid i] \\
&\quad - c_{k,j}\big\|\boldsymbol{B}^{(k)}[k\colon \mid k]\big\|\boldsymbol{e}_1 \\
&= \boldsymbol{A}[k\colon \mid j] + \Big(\sum_{i=1}^{k}\alpha_{i,j}^{(k+1)}\boldsymbol{A}[k\colon \mid i]\Big) - c_{k,j}\big\|\boldsymbol{B}^{(k)}[k\colon \mid k]\big\|\boldsymbol{e}_1 \\
\alpha_{i,j}^{(k+1)} &= \begin{cases} \alpha_{i,j}^{(k)} + c_{k,j}\alpha_{i,k}^{(k)} & i < k \\ c_{k,j} & i = k \end{cases}
\end{aligned}
$$

Finally, $\boldsymbol{B}^{(k+1)}[k+1\colon \mid j]$ directly follows, such that:

$$
\begin{aligned}
\boldsymbol{B}^{(k+1)}[k+1\colon \mid j] &= \boldsymbol{B}^{(k+1)}[k\colon \mid j][2:] \\
&= \Big(\boldsymbol{A}[k\colon \mid j] + \Big(\sum_{i=1}^{k}\alpha_{i,j}^{(k+1)}\boldsymbol{A}[k\colon \mid i]\Big) - c_{k,j}\big\|\boldsymbol{B}^{(k)}[k\colon \mid k]\big\|\boldsymbol{e}_1\Big)[2:] \\
&= \boldsymbol{A}[k+1\colon \mid j] + \Big(\sum_{i=1}^{k}\alpha_{i,j}^{(k+1)}\boldsymbol{A}[k+1\colon \mid i]\Big) - c_{k,j}\big\|\boldsymbol{B}^{(k)}[k\colon \mid k]\big\|\boldsymbol{e}_1[2:] \\
&= \boldsymbol{A}[k+1\colon \mid j] + \sum_{i=1}^{k}\alpha_{i,j}^{(k+1)}\boldsymbol{A}[k+1\colon \mid i]
\end{aligned}
$$

This concludes the proof by induction. Using above results we can compute $\boldsymbol{R}$ and $\boldsymbol{Q}$ efficiently. However, in practice we can generally avoid materialising $\boldsymbol{Q}$ and instead rely on applying the Householder reflections $\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots \boldsymbol{u}_n$ [40].

B.1.4 Explicit Expressions

Even though above shows that it is possible to compute $\boldsymbol{Q}$ and $\boldsymbol{R}$ in terms of efficiently computable inner-products, explicit expressions are necessary for an actual implementation. We provide such expressions here, using above notation and results.

$$\begin{aligned}
\|\boldsymbol{B}^{(k)}[k\colon|\,k]\| &= \sqrt{\sum_{i=1}^{k}\sum_{j=1}^{k}\alpha_{i,k}^{(k)}\alpha_{j,k}^{(k)}\langle \boldsymbol{A}[k\colon|\,i],\ \boldsymbol{A}[k\colon|\,j]\rangle} \\
\langle \boldsymbol{u}_k,\ \boldsymbol{B}^{(k)}[k\colon|\,t]\rangle &= \langle \boldsymbol{B}^{(k)}[k\colon|\,k],\ \boldsymbol{B}^{(k)}[k\colon|\,t]\rangle - \big\|\boldsymbol{B}^{(k)}[k\colon|\,k]\big\|\boldsymbol{B}^{(k)}[k,t] \\
&= \Bigg(\sum_{i=1}^{k}\alpha_{i,k}^{(k)}\Big(\langle \boldsymbol{A}[k\colon|\,i],\ \boldsymbol{A}[k\colon|\,t]\rangle + \sum_{j=1}^{k-1}\alpha_{j,t}^{(k)}\langle \boldsymbol{A}[k\colon|\,i],\ \boldsymbol{A}[k\colon|\,j]\rangle\Big)\Bigg) \\
&\qquad - \big\|\boldsymbol{B}^{(k)}[k\colon|\,k]\big\|\Big(a_{k,t} + \sum_{j=1}^{k-1}\alpha_{j,t}^{(k)}a_{k,j}\Big)
\end{aligned}$$

Finally, we add that for numerical stability (avoid catastrophic cancellation) one should in practice use:

$$\boldsymbol{u}_k := \boldsymbol{B}^{(k)}[k\colon|\,k] + \operatorname{sign}(\boldsymbol{B}^{(k)}[k,k])\big\|\boldsymbol{B}^{(k)}[k\colon|\,k]\big\|\boldsymbol{e}_1$$

B.2 A Final Word on Challenges

Above derivations constitute a (semi-)factorised rewrite of the Householder QR approach. Unfortunately, our approach requires some obscure notation which severely obfuscates the obtained expressions. Nevertheless, if one can look past the seemingly complex notation, the rewrite is not much more involved than that of factorised Gram Schmidt. However, there are some challenges associated to this particular rewrite.

First of all, this method requires the $n \times n$ submatrix of $\boldsymbol{A}$ where n is the number of features. In other words, it requires materialising the first n tuples of $\boldsymbol{A}$. Even though $n \ll m$ implies that this is computationally feasible, it is not in line with the principles of factorised databases.

More importantly, it is unclear whether the result is suitable for completely factorised applications. Whereas the $n \times n$ matrix $\boldsymbol{R}$ can be computed and materialised easily, computing the $m \times n$ matrix $\boldsymbol{Q}$ should be avoided at all cost. Clearly, this is no new observation and also applies to factorised Gram-Schmidt. The challenge is applying the factorised Householder reflections $\boldsymbol{u}_k$ without materialising $\boldsymbol{A}$.

We briefly outline this challenge by attempting to solve linear least squares using the factorised Householder QR result. Recall that as part of solving linear least squares we need to compute $\boldsymbol{Q}^T\boldsymbol{b}$, where $\boldsymbol{b}$ denotes the label vector. We obtain the following expressions:

$$
\begin{aligned}
\boldsymbol{b}^{(k+1)} &= \boldsymbol{Q}_k \boldsymbol{Q}_{k-1} \cdots \boldsymbol{Q}_1 \boldsymbol{b} \\
&= \boldsymbol{Q}_k \boldsymbol{b}^{(k)} \\
&= \begin{bmatrix} \boldsymbol{I}_{k-1} & \boldsymbol{0} \\ \boldsymbol{0} & \tilde{\boldsymbol{Q}}_k \end{bmatrix} \boldsymbol{b}^{(k)} \\
&= \begin{bmatrix} \boldsymbol{I}_{k-1} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{I} - c_k \boldsymbol{u}_k \boldsymbol{u}_k^\top \end{bmatrix} \begin{bmatrix} \boldsymbol{b}^{(k)}[1:k-1] \\ \boldsymbol{b}^{(k)}[k:m] \end{bmatrix} \\
&= \begin{bmatrix} \boldsymbol{b}^{(k)}[1:k-1] \\ \boldsymbol{b}^{(k)}[k:m] - c_k \langle \boldsymbol{u}_k,\ \boldsymbol{b}^{(k)}[k:m] \rangle \boldsymbol{u}_k \end{bmatrix} \\
\boldsymbol{Q}^\top \boldsymbol{b} &= \boldsymbol{Q}_n \boldsymbol{Q}_{n-1} \cdots \boldsymbol{Q}_1 \boldsymbol{b} = \boldsymbol{b}^{(n+1)}
\end{aligned}
$$

These expressions seem similar to expressions we have seen in the rewritten Householder QR method. Consequently, the computation of $\boldsymbol{Q}^\top \boldsymbol{b} = \boldsymbol{b}^{(n+1)}$ may allow a (second) rewrite to base terms. However, we have not been able to find an appropriate rewrite that does not require materialising $\boldsymbol{A}$. Nevertheless, even if such a rewrite exists, the computation of this alternative QR decomposition (the α-constants and $\boldsymbol{R}$) alone is significantly more involved and expensive than **F-GS**.