KroneDB

Compressing and Querying Time Series Data using the Kronecker Decomposition

Master's Thesis in Informatics

Thomas Rolf Mannhart
17-917-907

Department of Informatics of the University of Zurich
Prof. Dr. Dan Olteanu

Additional Supervisor: Dr. Johannes Marti

October 16, 2023

Abstract

This thesis introduces the design of *KroneDB*, a system for compressing time series data using the Kronecker decomposition, while allowing for efficient evaluation of relational queries including selection, projection, join, and aggregates (SPJA). KroneDB allows for a tunable trade-off between compression ratio and approximation error, while exploiting periodic patterns within the data to improve the compression. The compressed data can be queried directly without prior decompression while reducing the runtime of most queries. Updates can be applied directly to the compressed data and naturally enable value imputation and outlier detection in the updating process. By embedding our approach into the Functional Aggregate Queries (FAQ) framework, we show that it can be applied to a wide range of fundamental problems.

Zusammenfassung

Diese Arbeit präsentiert *KroneDB*, ein System zur Komprimierung von Zeitreihendaten unter Verwendung der Kronecker Decomposition. Es unterstützt dabei die effiziente Auswertung von relationalen Queries, welche Selection, Projection, Joins und Aggregates (SPJA) beinhalten können. Das Verhältnis zwischen Kompressionsrate und Approximationsfehler kann den Anforderungen entsprechend angepasst werden. Dabei werden periodische Muster in den Daten genutzt, um die Kompression zu verbessern. Queries können direkt auf den komprimierten Daten ausgeführt werden. Die Laufzeit ist dabei in der Regel kürzer als auf den unkomprimierten Daten. Updates lassen sich direkt auf die komprimierten Daten anwenden, wobei zusätzlich fehlende Werte ergänzt und Ausreisser erkannt werden. Das Einbetten unseres Ansatzes in das Functional Aggregate Queries (FAQ) Framework zeigt das Potential für eine Vielzahl weiterer Anwendungen auf.

Acknowledgments

I am grateful for the opportunity provided by Prof. Dr. Dan Olteanu and the University of Zurich to write this thesis. The fate and trust that Prof. Dr. Dan Olteanu put in me allowed me to develop and pursue my own ideas, while still providing guidance and support. I would also like to thank Dr. Johannes Marti, who jumped headfirst into a topic that was new to him and provided valuable feedback and ideas. My loved ones did not get enough attention from me during the last few months, but they were always there for me when I needed them and supported me in every way possible. I would especially like to thank my beloved Angela for pushing me to do my best and for her patience and understanding. The writing process was additionally supported by a long and beautiful summer in Zürich, my balcony, and my grill.

I also want to thank the providers of the publicly available datasets that I used in this thesis, MeteoSwiss [Met] and UTD19 [UTD], which allowed me to get confirmation that my ideas actually work.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Time series data is collected in many different fields, such as finance, logistics, meteorology, manufacturing, and many more. Whenever there are any sensors involved that measure anything over time, a time series is created. Think about sensors in machinery, meteorological stations, or traffic counters. The purpose of collecting this data is to use it for further analysis such as forecasting or anomaly detection.

Time series data is usually stored in specialized time series databases, which are optimized for storing and querying time series data [DF14]. A major component of these optimizations is data compression. The field of data compression in databases is very broad but mainly focuses on compressing sparse data with lossless compression techniques such as run-length encodings [LMF+16, AMF06, RVH93], frame of Reference encoding [LMF+16], dictionary compression [LMF+16, AMF06], delta encoding [RVH93], and null suppression [AMF06, RVH93].

A widely used lossy compression technique in time series databases is to compress clusters of continuous data points by sampling a single point from or taking the average over the cluster. Some more advanced techniques based on Principal Component Analysis (PCA) or Singular Value Decomposition (SVD) are less commonly used [Fu11].

This thesis introduces a system called *KroneDB* which uses the Kronecker decomposition [Loa00] to get a lossy compression of dense time series data with a tunable compression ratio and approximation error. The Kronecker decomposition is especially well suited for time series that contain periodic patterns because this structural information can be used to improve the compression of the data. Furthermore, the Kronecker decomposition allows for multiple similar time series to be compressed together, which further improves the compression rate.

In addition to the efficiency gains in storage, the Kronecker decomposition also allows for efficient aggregations, like calculating the sum or sum of products of features directly over the compressed data. These aggregations are very common in the analysis of time series data and can be combined to calculate more complex metrics, like the variance, standard deviation, or correlations. Relation queries including selections, projections, joins, and aggregations (SPJA) are supported and can be executed directly on the compressed data.

KroneDB levarages traditional relational databases to store the compressed data and to query it efficiently using SQL. This avoids the need for specialized time series databases.

Additionally, KroneDB provides update methods to efficiently update the compressed data when new data points are added to the time series. This is especially important in a streaming scenario, where new data points are added to the time series in real-time. Not only can the new data points be added to the compressed data very efficiently, but also, can gaps in the new data be filled and outliers can be detected and removed, with little to no overhead.

Our approach is also discussed within the context of the Functional Aggregate Query (FAQ) framework [AKNR16]. By showing how our work integrates naturally with FAQ, we effectively enable its use across a wide range of fundamental problems beyond query evaluation in databases including constraint satisfaction problems, linear algebra (matrix chain multiplication, discrete Fourier transform), satisfiability, inference and learning in probabilistic graphical models, machine

learning (cost functions, covariance matrices, learning), and tensor networks in physics.

1.1 Motivation

This work is motivated by the increasing amount of time series data that is collected, stored, and analyzed. To store and analyze this data, specialized time series databases are used which compress the data by sampling and averaging older data points and providing specialized data structures to efficiently process time series data. In general, dense data is not well supported by relational databases and is usually stored in specialized databases or extracted into specialized analysis and machine learning tools. The Kronecker decomposition is a promising approach to compress dense data to store and query it in relational databases. This allows us to avoid the need for many specialized systems and to use the very well-established and highly optimized relational database management systems (RDBMSs) instead.

1.2 Contributions

A system called *KroneDB* is introduced that implements these methods and can be used to compress, query, and update time series data using any RDBMS like DuckDB [RM19] or a custom implementation using Python and NumPy.

The main contribution of this thesis is an investigation of the methods that can be used to compress, query, and update time series data using the Kronecker decomposition.

- The many choices that need to be made when using the Kronecker decomposition for a specific dataset are presented and discussed and their effects are shown directly on real-world data.
- Given a query including selection, projection, join, and aggregates (SPJA) and a relational database, where some of the relations are decomposed using Kronecker decomposition, we show how to rewrite the original query to observe the decomposition and thereby allow for more efficient execution.
- We introduce an efficient approach to update a decomposed relation in the compressed domain. Our approach also naturally allows for value imputation and outlier detection.
- The query rewriting and optimization are additionally shown using the FAQ framework, which allows for easy inclusion of decomposed relations into larger and more complex queries and shows that this approach is not limited to SPJA queries but can be extended to many more fundamental problems.
- The compression ratio and approximation error of the Kronecker decomposition are evaluated on real-world data and it can be shown that for temperature measurements, we achieve a compression ratio of 1:1500 with a root mean square error (RMSE) of 3°C and a compression ratio of 1:300 with an RMSE of just 1.5°C for a temperature range of 57.3°C.
- The runtime of the queries evaluated on real-world data shows a 1.3x to 2.3x speedup at a 0.0% to 0.7% deviation from the real result for the sum and a 1.4x to 2.8x speedup at a 0.5% to 3% deviation for the sum product when executed using DuckDB.

1.3 Outline

The rest of this thesis is structured as follows: Chapter 2 will introduce the Kronecker decomposition and the FAQ framework in detail. The architecture of KroneDB together with a motivating example is presented in Chapter 3. The analysis of all the different choices for the Kronecker decomposition and their effects on the compressed data is presented in Chapter 4. How queries need

to be rewritten and optimized to work with the compressed data is shown in Chapter 5 together with experimental results on real-world data using KroneDB to execute the queries. Updating together with value imputation and outlier detection is presented in Chapter 6. How KroneDB fits into the FAQ framework is shown in Chapter 7. Related work to time series databases and the Kronecker decomposition is introduced in Chapter 8. In Chapter 9, some open ends are discussed and possible future work is presented. Finally, Chapter 10 contains the conclusion of this thesis.

Chapter 2

Preliminaries

In this chapter, we introduce the most important concepts, notations, and naming conventions, that are used throughout this thesis.

In Section 2.1, we introduce the Kronecker decomposition, showing that it is a kind of low-rank approximation and explain how it can be constructed using the singular value decomposition. In Section 2.2, we introduce the framework of Functional Aggregate Queries (FAQ), discuss how we can represent database queries as FAQs, and how this representation can be used to optimize the query evaluation.

2.1 Kronecker Decomposition

In this section, we discuss how a matrix can be decomposed into a sum of Kronecker products. We start by defining the Kronecker Product. Then we show how a matrix can be decomposed using Low-Rank Approximation [RV22] with singular value decomposition (SVD) [KL80], and finally how to use the Low-Rank Approximation to get the Kronecker decomposition [Loa00].

2.1.1 Kronecker Product

Definition 1. Given two matrices $S \in \mathbb{R}^{m_s \times n_s}$ and $P \in \mathbb{R}^{m_p \times n_p}$, the result of their Kronecker product $S \otimes P$ is defined as an $m_s \times n_s$ block matrix, where the block (i, j) is the product $s_{ij}P$ [Loa00]. To keep the terminology consistent throughout the thesis, we call S the *scaling-matrix*, P the *period-matrix* and the result of the Kronecker product $S \otimes P$ the *data-matrix* $D \in \mathbb{R}^{m_d \times n_d}$, where $m_d = m_s m_p$ and $n_d = n_s n_p$.

Example 1. Consider the following two matrices $S \in \mathbb{R}^{3\times 2}$ and $P \in \mathbb{R}^{2\times 2}$:

$$S = \begin{bmatrix} s_{11} & s_{12} \\ s_{21} & s_{22} \\ s_{31} & s_{32} \end{bmatrix} \text{ and } P = \begin{bmatrix} p_{11} & p_{12} \\ p_{21} & p_{22} \end{bmatrix}.$$

Then, $S \otimes P$ is defined as

$$\begin{bmatrix} s_{11}P & s_{12}P \\ s_{21}P & s_{22}P \\ s_{31}P & s_{32}P \end{bmatrix} = \left[\begin{array}{cc|cc} s_{11}p_{11} & s_{11}p_{12} & s_{12}p_{11} & s_{12}p_{12} \\ s_{11}p_{21} & s_{11}p_{22} & s_{12}p_{21} & s_{12}p_{22} \\ \hline s_{21}p_{11} & s_{21}p_{12} & s_{22}p_{11} & s_{22}p_{12} \\ s_{21}p_{21} & s_{21}p_{22} & s_{22}p_{21} & s_{22}p_{22} \\ \hline s_{31}p_{11} & s_{31}p_{12} & s_{32}p_{11} & s_{32}p_{12} \\ s_{31}p_{21} & s_{31}p_{22} & s_{32}p_{21} & s_{32}p_{22} \end{array}\right] = D,$$

where $D \in \mathbb{R}^{6\times 4}$.

Another definition of the Kronecker product is that every element d_{ij} of the result matrix $D \in \mathbb{R}^{m_d \times n_d}$ is defined as

$$d_{ij} = s_{i_s j_s} \cdot p_{i_p j_p}, \tag{2.1}$$

where $s_{i_s j_s}$ and $p_{i_p j_p}$ are elements of the scaling and period matrices $S \in \mathbb{R}^{m_s \times n_s}$ and $P \in \mathbb{R}^{m_p \times n_p}$ respectively, such that

$$i = (i_s - 1) \cdot m_p + i_p \quad \text{and} \quad j = (j_s - 1) \cdot n_p + j_p, \tag{2.2}$$

where $i \in [m_d]$, $j \in [n_d]$, $i_s \in [m_s]$, $j_s \in [n_s]$, $i_p \in [m_p]$, and $j_p \in [n_p]$. This definition will be used throughout the thesis.

2.1.2 Low-Rank Approximation

The goal of low-rank approximation is to find the best rank-k matrix D_k to approximate the data-matrix $D \in \mathbb{R}^{m_d \times n_d}$, where $k < \min\{m_d, n_d\}$. The matrix rank is the largest number of linearly independent rows or columns in a matrix. [RV22]

The best approximation can be found using the singular value decomposition (SVD) of the matrix $D \in \mathbb{R}^{m_d \times n_d}$, which is defined as $D = U\Sigma V^\top$, where $U \in \mathbb{R}^{m_d \times m_d}$ is the orthogonal matrix whose columns are the left singular vectors, $\Sigma \in \mathbb{R}^{m_d \times n_d}$ is the diagonal matrix of singular values and $V \in \mathbb{R}^{n_d \times n_d}$ is the orthogonal matrix whose columns are the right singular vectors [KL80]. The singular values are ordered on the diagonal of Σ such that $\sigma_{11} \geq \sigma_{22} \geq \cdots \geq \sigma_{\min\{m_d,n_d\}\min\{m_d,n_d\}} \geq 0$. To construct D_k we use the first k singular values and vectors:

$$D_k = U[:, :k]\Sigma[:k, :k]V[:, :k]^\top,$$

where $U[:, :k] \in \mathbb{R}^{m_d \times k}$ are the first k columns of U, $\Sigma[:k, :k] \in \mathbb{R}^{k \times k}$ are the first k rows and columns of Σ, and $V[:, :k] \in \mathbb{R}^{n_d \times k}$ are the first k columns of V. Using this construction, it holds that

$$\|D - D_k\|_F \leq \|D - X_k\|_F$$

for every rank-k matrix $X_k \in \mathbb{R}^{m_d \times n_d}$, where $\|\cdot\|_F$ is the Frobenius norm. [RV22]

The low-rank approximation can be used to compress, de-noise, or complete a matrix [RV22]. In this thesis, in Chapter 4 we will focus on the compression and observe the de-noising effect. The completion capabilities will be used in Section 6.2.

2.1.3 A Low-Rank Kronecker Product Decomposition

Definition 2. The Kronecker decomposition is a low-rank approximation, where the goal is to find the best rank-k approximation $D_k \in \mathbb{R}^{m_d \times n_d}$ of the data-matrix $D \in \mathbb{R}^{m_d \times n_d}$ such that

$$D_k = \sum_{r=1}^{k} S^r \otimes P^r, \tag{2.3}$$

where $S^r \in \mathbb{R}^{m_s \times n_s}$ and $P^r \in \mathbb{R}^{m_p \times n_p}$ are the scaling and period matrices for the rth rank of the decomposition, $n_d = n_s n_p$, and $m_d = m_s m_p$.

Example 2. Consider a data-matrix $D \in \mathbb{R}^{6\times 4}$, which is decomposed into a sum of Kronecker products, such that $S^r \in \mathbb{R}^{3\times 2}$ and $P^r \in \mathbb{R}^{2\times 2}$ for all $r \in [k]$ such that,

$$S^r = \begin{bmatrix} s^r_{11} & s^r_{12} \\ s^r_{21} & s^r_{22} \\ s^r_{31} & s^r_{32} \end{bmatrix} \text{ and } P^r = \begin{bmatrix} p^r_{11} & p^r_{12} \\ p^r_{21} & p^r_{22} \end{bmatrix}.$$

Then, the Kronecker decomposition of rank-k is defined as

$$D_k = \left[\begin{array}{c|c} s_{11}^1 P^1 + \cdots + s_{11}^k P^k & s_{12}^1 P^1 + \cdots + s_{12}^k P^k \\ \hline s_{21}^1 P^1 + \cdots + s_{21}^k P^k & s_{22}^1 P^1 + \cdots + s_{22}^k P^k \\ \hline s_{31}^1 P^1 + \cdots + s_{31}^k P^k & s_{32}^1 P^1 + \cdots + s_{32}^k P^k \end{array}\right] = \sum_{r=1}^{k} S^r \otimes P^r,$$

where s_{ij}^r and p_{ij}^r are the elements of the scaling and period matrices for the rth rank of the decomposition.

To achieve this goal, the construction of D_k from Section 2.1.2 is slightly modified. Before computing the SVD, we reshape the data-matrix $D \in \mathbb{R}^{m_s m_p \times n_s n_p}$ into a matrix $\mathcal{R}(D) \in \mathbb{R}^{m_s n_s \times m_p n_p}$, by vectorizing each block into a column vector and stacking the transposed column vectors into a matrix. Therefore,

$$\mathcal{R}\left(\begin{bmatrix} s_{11}P & s_{12}P \\ s_{21}P & s_{22}P \\ s_{31}P & s_{32}P \end{bmatrix}\right) = \begin{bmatrix} vec(s_{11}P)^\top \\ vec(s_{21}P)^\top \\ vec(s_{31}P)^\top \\ vec(s_{12}P)^\top \\ vec(s_{22}P)^\top \\ vec(s_{32}P)^\top \end{bmatrix} = \left[\begin{array}{cccc} s_{11}p_{11} & s_{11}p_{21} & s_{11}p_{12} & s_{11}p_{22} \\ \hline s_{21}p_{11} & s_{21}p_{21} & s_{21}p_{12} & s_{21}p_{22} \\ \hline s_{31}p_{11} & s_{31}p_{21} & s_{31}p_{12} & s_{31}p_{22} \\ \hline s_{12}p_{11} & s_{12}p_{21} & s_{12}p_{12} & s_{12}p_{22} \\ \hline s_{22}p_{11} & s_{22}p_{21} & s_{22}p_{12} & s_{22}p_{22} \\ \hline s_{32}p_{11} & s_{32}p_{21} & s_{32}p_{12} & s_{32}p_{22} \end{array}\right].$$

The SVD $\mathcal{R}(D) = U\Sigma V^\top$, is used to construct the Kronecker decomposition D_k as

$$D_k = \sum_{r=1}^{k} (vec^{-1}_{m_s,n_s}(U[:,r]) \cdot \Sigma[r,r]) \otimes vec^{-1}_{m_p,n_p}(V[:,r]) = \sum_{r=1}^{k} S^r \otimes P^r, \tag{2.4}$$

where $vec^{-1}_{m,n}$ is the inverse of the vec operator, which reshapes a column vector of length mn into a matrix of size $m \times n$ by dividing it into n vectors of size m and stacking them horizontally. Thus,

$$vec^{-1}_{3,2}\left(\begin{bmatrix} s_{11} \\ s_{21} \\ s_{31} \\ s_{12} \\ s_{22} \\ s_{32} \end{bmatrix}\right) = \left[\begin{array}{c|c} s_{11} & s_{12} \\ s_{21} & s_{22} \\ s_{31} & s_{32} \end{array}\right].$$

The rank-wise scaling and period matrices S^r and P^r are constructed as follows:

$$S^r = vec^{-1}_{m_s,n_s}(U[:,r]) \cdot \Sigma[r,r] \quad \text{and} \quad P^r = vec^{-1}_{m_p,n_p}(V[:,r]).$$

The scalar factor $\Sigma[r,r]$ can be freely distributed into the scaling and period matrices S^r and P^r. In fact, if S_r and P_r are optimal, then it holds that αS_r and $(1/\alpha)P_r$ are also optimal for every $\alpha \neq 0$. We generally choose to distribute the scalar factor $\Sigma[r,r]$ into the scaling-matrix S^r. This brings the advantage, that the ranks of the period matrix are orthonormal, which will be useful in Chapter 6. Using the construction from (2.4), it holds that

$$\|D - D_k\|_F \leq \|D - X_k\|_F$$

for every matrix $X_k \in \mathbb{R}^{m_d \times n_d}$ that is the sum of k Kronecker products. [Loa00]

2.2 Functional Aggregate Queries: FAQ

In Section 7, the Functional Aggregate Query (FAQ) framework is used to represent and evaluate KroneDB queries. The FAQ framework was introduced by Abo Khamis et al. [AKNR16]. It gives

syntax and semantics for expressing frequently asked questions across computer science, such as database queries, graph reachability, and boolean satisfiability. There are efficient algorithms developed for solving FAQ expressions, e.g. the InsideOut algorithm.

An FAQ expression has the form

$$\phi(\mathbf{x}_f) = \bigoplus_{x_{f+1} \in Dom(X_{f+1})}^{(f+1)} \cdots \bigoplus_{x_n \in Dom(X_n)}^{(n)} \bigodot_{S \in \mathcal{E}} \psi_S(\mathbf{x}_S),$$

where $\mathbf{x}_f$ is the tuple of *free* variables $x_1, \ldots, x_f$ and $x_{f+1}, \ldots, x_n$ are the *bound* variables. For every variable x_i there is a fixed set $Dom(X_i)$ that is the domain of x_i. The (multi)set $\mathcal{E}$ is a set of hyperedges in a (multi)hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of vertices $1, \ldots, n$. Every hyperedge $S \in \mathcal{E}$ is a subset of the vertices $S \subseteq \mathcal{V}$ and has an associated input factor $\psi_S(\mathbf{x}_S)$. The input factors are functions that map tuples from $\prod_{i \in S} Dom(X_i)$ to elements in the domain $\mathbb{D}$, of a finite semiring $(\mathbb{D}, \oplus, \odot, \mathbf{0}, \mathbf{1})$. In the semiring $(\mathbb{D}, \oplus, \odot, \mathbf{0}, \mathbf{1})$, $\oplus$ and $\odot$ denote the sum and product operators, and $\mathbf{0} \in \mathbb{D}$ and $\mathbf{1} \in \mathbb{D}$ are the additive and multiplicative identities for $\oplus$ and $\odot$, respectively. [AKNR16]

Database queries are represented as FAQ expressions by using the relations as hyperedges in $\mathcal{E}$ and the attributes as variables in $\mathcal{V}$ of the hypergraph $\mathcal{H}$.

Consider the relation **R**

X_1	X_2	X_3	X_4
a	a	0	1
a	a	1	1
a	b	0	1
b	b	1	0

where X_1 and X_2 are categorical over {'a', 'b'} and X_3 and X_4 are binary. Using the FAQ framework, we can represent this relation as a factor

$$\psi_R(x_1, x_2, x_3, x_4) = \begin{cases} \mathbf{1} & \text{if the tuple } (x_1, x_2, x_3, x_4) \text{ appears in } \mathbf{R}, \\ \mathbf{0} & \text{otherwise,} \end{cases}$$

where **1** and **0** are the multiplicative and additive identities of the semiring. This means that only tuples mapped to **1** are stored in the relation and all other tuples are not. This representation where the factors only return **1** or **0** can always be used without loss of generality by adding virtual factors that return the actual values of the attributes, as will be shown in the examples below.

FAQ Examples

To explain how a database query is represented as an FAQ, we will provide a few examples, starting with a projection query.

Projection Consider the query

```
SELECT X_1, X_3 FROM R;
```

which we want to represent as an FAQ. We can do this by using the sum-product semiring $(\mathbb{N}, +, \cdot, 0, 1)$, where $\mathbb{N}$ is the set of natural numbers including 0, to get the FAQ

$$\phi_1(x_1, x_3) = \sum_{x_2, x_4} \psi_R(x_1, x_2, x_3, x_4),$$

where ϕ_1 is a function that maps the free variables x_1 and x_3 to the number of tuples in **R** that have the values x_1 and x_3 in the attributes X_1 and X_3, respectively. The domains of the attributes

are $Dom(X_1) = Dom(X_2) = \{\text{'a'}, \text{'b'}\}$ and $Dom(X_3) = Dom(X_4) = \{0, 1\}$. The domain of ϕ_1 is the cross-product of the domains of the free variables, therefore,

$$Dom(\phi_1) = Dom(X_1) \times Dom(X_3) = \{\text{'a'}, \text{'b'}\} \times \{0, 1\} = \{(\text{'a'}, 0), (\text{'a'}, 1), (\text{'b'}, 0), (\text{'b'}, 1)\}.$$

We pass each tuple in $Dom(\phi_1)$ to ϕ_1, starting with $(\text{'a'}, 0)$. The return values of the factor $\psi_R(\text{'a'}, x_2, 0, x_4)$ for every tuple (x_2, x_4) in $Dom(X_2) \times Dom(X_4)$ are

$$\begin{aligned}
\psi_R(\text{'a'}, \text{'a'}, 0, 0) &= 0, \\
\psi_R(\text{'a'}, \text{'a'}, 0, 1) &= 1, \\
\psi_R(\text{'a'}, \text{'b'}, 0, 0) &= 0, \\
\psi_R(\text{'a'}, \text{'b'}, 0, 1) &= 1.
\end{aligned}$$

The individual results are summed up to get $\phi_1(\text{'a'}, 0) = 2$. The same process is repeated for $\phi_1(\text{'a'}, 1)$, $\phi_1(\text{'b'}, 0)$ and $\phi_1(\text{'b'}, 1)$, which results in

$$\begin{aligned}
\phi_1(\text{'a'}, 0) &= 2, \\
\phi_1(\text{'a'}, 1) &= 1, \\
\phi_1(\text{'b'}, 0) &= 0, \\
\phi_1(\text{'b'}, 1) &= 1.
\end{aligned}$$

The final result relation $\mathbf{T}_1$ is then constructed as

X_1	X_3
a	0
a	0
a	1
b	1

.

Join For the second example, consider a second relation **S**

X_1	X_5
a	0
a	1
b	1

.

The join of **R** and **S** with a projection on X_1, X_3 and X_5 is expressed in SQL as

```
SELECT R.X_1, R.X_3, S.X_5 FROM R JOIN S ON R.X_1 = S.X_1;
```

and in FAQ over the sum-product semiring as

$$\phi_2(x_1, x_3, x_5) = \sum_{x_2, x_4} \psi_R(x_1, x_2, x_3, x_4) \cdot \psi_S(x_1, x_5)$$

with the domain

$$Dom(\phi_2) = Dom(X_1) \times Dom(X_3) \times Dom(X_5) = \{(\text{'a'}, 0, 0), (\text{'a'}, 0, 1), \ldots, (\text{'b'}, 1, 0), (\text{'b'}, 1, 1)\}.$$

The results for the input tuples are as follows:

$$\begin{aligned}
&\phi_2(\text{'a'}, 0, 0) = 2,\\
&\phi_2(\text{'a'}, 0, 1) = 2,\\
&\vdots\\
&\phi_2(\text{'b'}, 1, 0) = 0,\\
&\phi_2(\text{'b'}, 1, 1) = 1.
\end{aligned}$$

The result relation $\mathbf{T}_2$ is then constructed as

X_1	X_3	X_5
a	0	0
a	0	0
a	0	1
a	0	1
a	1	0
a	1	1
b	1	1

.

Aggregation The last example is a join of $\mathbf{R}$ and $\mathbf{S}$ with a `SUM` aggregation on $X_3 * X_5$ and a projection on X_1. This is expressed in SQL as

```
SELECT R.X_1, SUM(R.X_3 * S.X_5)
FROM R JOIN S ON R.X_1 = S.X_1
GROUP BY R.X_1;
```

and in FAQ over the sum-product semiring as

$$\phi_3(x_1) = \sum_{x_2,x_3,x_4,x_5} \psi_R(x_1, x_2, x_3, x_4) \cdot x_3 \cdot \psi_S(x_1, x_5) \cdot x_5.$$

The variables x_3 and x_5, that stand outside of any factor are shorthands for the virtual factors

$$\psi_{x_3}(x_3) = x_3 \quad \text{and} \quad \psi_{x_5}(x_5) = x_5,$$

respectively. The function ϕ_4 maps the free variable x_1 to the sum of the products of the values of X_3 and X_5. It does not return the number of tuples in the result relation, but the result of the aggregation for each group:

$$\begin{aligned}
&\phi_3(\text{'a'}) = 1,\\
&\phi_3(\text{'b'}) = 1.
\end{aligned}$$

The query result $\mathbf{T}_3$ is then constructed as

X_1	SUM($X_3 * X_5$)
a	1
b	1

.

Chapter 3

KroneDB - An Overview

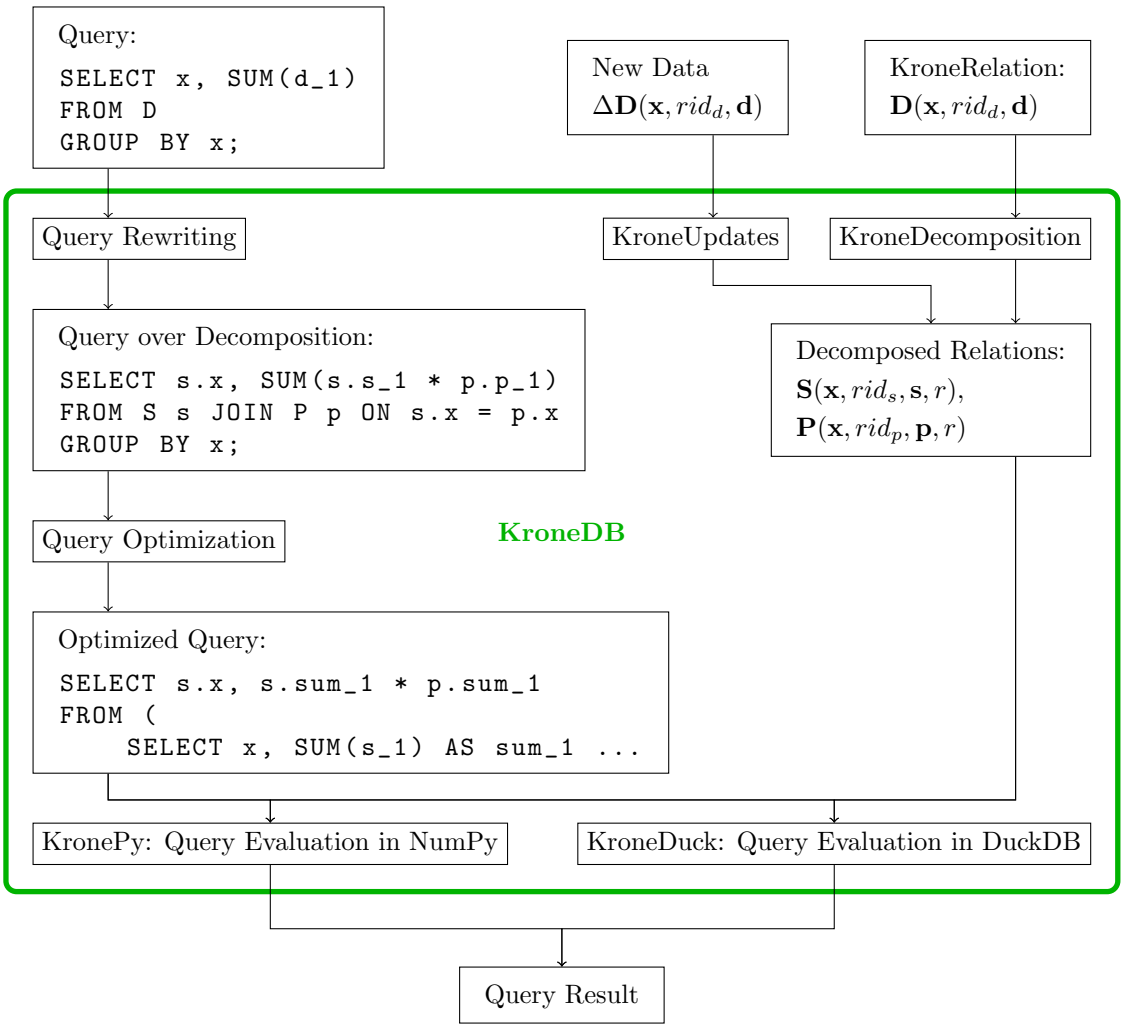


Figure 3.1: KroneDB Architecture

KroneDB is a database system that uses the Kronecker decomposition to speed up query evaluation and save storage space. The greatest strength of KroneDB is the compression and efficient querying of time series data, like sensor readings. It can however also be used for any other kind

of dense numerical data. It is implemented in Python and can use NumPy and DuckDB for query evaluation. The architecture of KroneDB is shown in Figure 3.1. The user provides a query and a KroneRelation. A KroneRelation is a relation whose columns are strictly separated into key columns, $\mathbf{x}$, and value columns, $\mathbf{d}$. More details about the KroneRelation are given in Section 4.1. The KroneRelation is then decomposed using KroneDecomposition, which is a Kronecker decomposition on the value columns of the KroneRelation. How the KroneDecomposition works and which parameters can be adjusted are all explained in Chapter 4. The query processing including rewriting and optimization is explained in Chapter 5. The optimized query is then evaluated using KronePy or KroneDuck, where KronePy provides a Python API to evaluate specific queries using NumPy arrays and KroneDuck generates SQL queries and evaluates them using DuckDB. It is also possible to update the Decomposed KroneRelations directly using KroneUpdates. The update process additionally allows the approximation of missing values and the detection of outliers. The outliers can be corrected automatically by replacing them with the approximation. KroneUpdates are discussed in Chapter 6. Chapter 7 discusses another possibility to represent, rewrite, and evaluate KroneDB queries using the FAQ framework.

Motivating Example

In this section, we want to motivate the use of the Kronecker decomposition as a compression method. We will do this by applying the Kronecker decomposition to one particular real-world time series. We will show how the Kronecker decomposition achieves a better compression ratio and a better reconstruction accuracy compared to averaging methods, which are commonly used for the compression of time series.

stn	time	temperature (°C)
KLO	199001010000	-3.5
KLO	199001010010	-3.4
⋮	⋮	⋮
KLO	202308230420	20.0
KLO	202308230430	19.9

Table 3.1: The current temperature 2 meters above ground in Kloten ZH measured every 10 minutes in °C between 01.01.1990 and 23.08.2023. *Data Source: MeteoSwiss [Met].*

Consider a meteorological station that measures the current temperature every 10 minutes as shown in Table 3.1. We want to compress this data to reduce its size on disk while still being able to reconstruct the original data. Lossless compression methods such as Run-length Encoding or Bit-packing [Enc] achieve this by removing redundant information in the data.

Plotting a slice of our example dataset in Figure 3.2 reveals an obvious and intuitive pattern in the data. The temperature rises during the day and falls during the night. We expect the temperature to follow this daily pattern, which we call the daily period of the time series. Therefore, it should be possible to approximate the temperature along an arbitrary day by using the daily period and fitting it to the average temperature of this day. How this could look like is shown in Figure 3.3. Note that we purposefully ignore the fact, that negative temperature values would flip the daily pattern and will come back to this in Section 4.4.

Comparing Figure 3.3 to the Kronecker decomposition of a single column in Figure 3.4, shows a similarity between the two. The period-matrix P of the Kronecker decomposition corresponds to the typical day and the scaling-matrix corresponds to the average temperature for each day. This suggests that the Kronecker decomposition might be a good method to compress the temperature data.

To apply the Kronecker decomposition to the temperature data, we first need to decide on the height of the period-matrix. The natural daily period in the data contains 144 10-minute

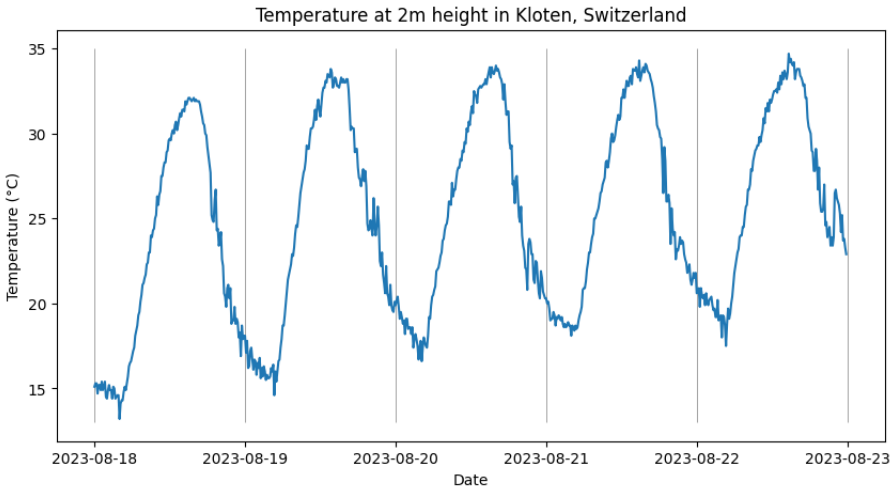


Figure 3.2: The last five days of our time series. The ticks on the Date axis are at 00:00 of the indicated day. *Data Source: MeteoSwiss.*

$$\left[\begin{array}{l|l} \text{Day 1 06:00} & 12°\text{C} \\ \text{Day 1 14:00} & 18°\text{C} \\ \text{Day 1 22:00} & 15°\text{C} \\ \text{Day 2 06:00} & 14°\text{C} \\ \text{Day 2 14:00} & 22°\text{C} \\ \text{Day 2 22:00} & 18°\text{C} \\ \text{Day 3 06:00} & 16°\text{C} \\ \text{Day 3 14:00} & 24°\text{C} \\ \text{Day 3 22:00} & 20°\text{C} \end{array}\right] = \left[\begin{array}{l|l} \text{Day 1} & 15°\text{C} \cdot \text{TD} \\ & \\ & \\ \text{Day 2} & 18°\text{C} \cdot \text{TD} \\ & \\ & \\ \text{Day 3} & 20°\text{C} \cdot \text{TD} \\ & \end{array}\right], \text{where TD} = \left[\begin{array}{l|l} 06{:}00 & 0.8 \\ 14{:}00 & 1.2 \\ 22{:}00 & 1.0 \end{array}\right]$$

Figure 3.3: Example showing how we can encode the temperature measurements over three days using a typical day (TD) and the average temperature for each day. It is assumed that on a typical day, at 06:00 in the morning, the temperature is 20% lower, and at 14:00, the temperature is 20% higher than the average. At 22:00, the temperature is assumed to be equal to the average.

$$\begin{bmatrix} s_{11} \\ s_{21} \\ s_{31} \end{bmatrix} \otimes \begin{bmatrix} p_{11} \\ p_{21} \\ p_{31} \end{bmatrix} = \begin{bmatrix} s_{11}p_{11} \\ s_{11}p_{21} \\ s_{11}p_{31} \\ s_{21}p_{11} \\ s_{21}p_{21} \\ s_{21}p_{31} \\ s_{31}p_{11} \\ s_{31}p_{21} \\ s_{31}p_{31} \end{bmatrix} = \begin{bmatrix} s_{11}P \\ \\ \\ s_{21}P \\ \\ \\ s_{31}P \\ \\ \end{bmatrix}, \text{where } P = \begin{bmatrix} p_{11} \\ p_{21} \\ p_{31} \end{bmatrix}$$

Figure 3.4: Example showing how the period-matrix is repeated if we apply Kronecker decomposition to a single-column matrix.

intervals, therefore we choose the height of the period-matrix as $m_p = 144$. To be able to divide the time series into complete days, we remove the last, unfinished day from the data, such that $m_d \,\%\, m_p = 0$, where m_d is the height of the data matrix, i.e. the number of data points in the time series. The modulo operator % returns the remainder of the division. We end up with $m_{data} = 1'769'328$ tuples of data, which corresponds to 12'287 complete days or more than 33 years. The result of the Kronecker decomposition is shown in Figure 3.5. Note that we cleaned the data by interpolating any missing or erroneous measurements using linear interpolation, before applying the Kronecker decomposition.

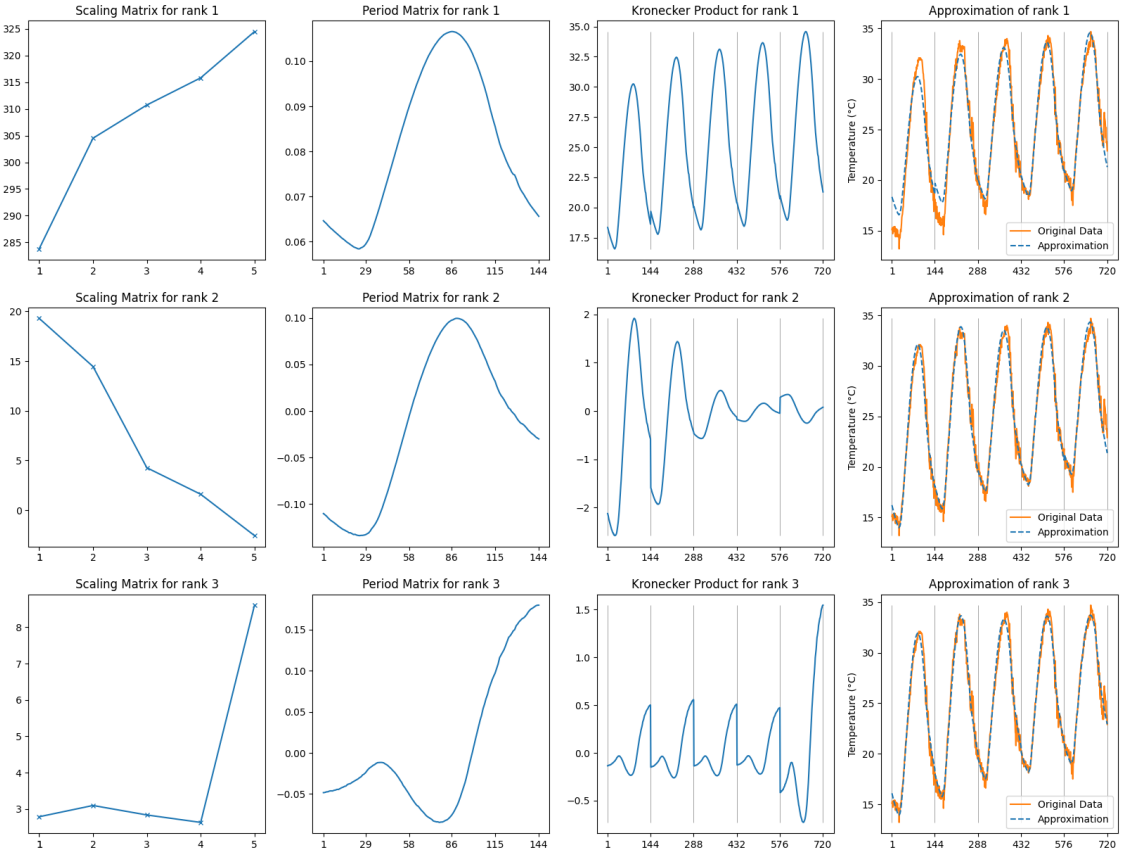


Figure 3.5: The result of applying Kronecker decomposition to the temperature data. For ranks 1, 2, and 3, we show the last five days of the reconstructed data. the scaling- and period-matrices are shown in the first two columns. The third column shows the Kronecker product $S^r \otimes R^r$ for the specific rank r, as described in (2.3). The fourth column shows the reconstructed data up to the specified rank, i.e. D_1, D_2, and D_3, compared to the original data.

Looking at rank 1, the scaling-matrix follows the general trend over the days. The scaling values rise from the first to the last day, which corresponds to the temperature rising over the days. Note that the scaling does not contain the actual average temperatures, but scaling it by a factor $\alpha \approx 1/13$, as discussed in Section 2.1.3, would result in values close to the average temperatures. The period-matrix in the second column captures the daily period, as expected. It starts with a downward slope, which corresponds to the temperature falling during the night. Then it rises, which corresponds to the temperature rising during the day and falls again in the evening. Comparing the approximation to the original data shows that the reconstructed data follows the general trend of the original data, but the accuracy varies for the different days. The temperature range over the first day especially seems to be compressed compared to the original data. The

same phenomenon can be observed for the second and third days, but it is less pronounced. For the fifth day, we can observe the opposite effect, where the approximated temperatures are stretched compared to the original data. The reason for this phenomenon is that the scaling does not only shift the period-matrix but also stretches it. This is an advantage for the temperature data because the temperature range is not the same for every day and is in general larger for warmer days.

The expectation from the low-rank decomposition is that the greatest variation in the data is captured by the first rank and that the second rank should be comparatively small in absolute values. Comparing the Kronecker products in the third column confirms this expectation. The second rank can be seen as a correction to the first rank, where specific days divert from the norm. It therefore influences certain days more than others, which can be seen in the scaling-matrix. This correction is encoded in the period-matrix and looks again like the daily period, but it is roughly centered around 0. This means that applying this correction to a certain day will stretch the daily period if the scaling is positive and compress it if the scaling is negative. We can see that this is the case for the reconstructed data, where the first days are stretched and the last day is compressed. This correction seems to fix the major limitations of the rank 1 reconstruction.

Finally, rank 3 is supposed to capture even less variation than rank 2, which is why the range of the Kronecker Product is even smaller. Looking at the scaling-matrix, we can see that this correction mainly applies to the fifth day, where the scaling is much larger than for the other days. The period-matrix is again a correction that is roughly centered around 0 and it seems to suppress the peak at the hottest time of the day but more prominently it raises the temperature in the evening. This can be seen in the reconstructed data, where the temperature stays higher for longer in the evening of the fifth day.

With three ranks, the approximation of the temperature data is already very good. To show how good the approximation is, we can calculate the root mean square error (RMSE) of the reconstruction. The Kronecker decomposition is compared to two other methods, simple averaging, and averaging with linear interpolation. The simple averaging method divides the data into patches of a certain size and takes the average of each patch. For the approximation, we reconstruct each datapoint by using the average of the patch that contains the datapoint. The averaging with linear interpolation method uses the same average but interpolates the average between two patches linearly to reconstruct the data points between the patch centers.

Kronecker decomposition			Averaging		
Rank	CR	RMSE (°C)	CR	RMSE Rep. (°C)	RMSE Int. (°C)
1	142.3	2.06	144.0	3.28	3.21
2	71.2	1.19	72.0	2.73	2.73
3	47.5	0.89	48.0	1.83	1.57

Table 3.2: Comparison of the root mean square error (RMSE) in °C of the Kronecker decomposition and two averaging methods. For the comparison we selected for each rank of the Kronecker decomposition a patch size for the averaging methods that results in a compression ratio (CR) that is as close as possible to the compression ratio of the Kronecker decomposition. The slightly larger compression ratio of the averaging methods is because the Kronecker decomposition stores the period-matrix in addition to the scaling-matrix. This difference is negligible for the comparison.

In Table 3.2, we compare the RMSE of the Kronecker decomposition to the RMSE of the simple averaging and averaging with linear interpolation. The compression ratio of the averaging method corresponds to the patch size, e.g. a patch size of 144 corresponds to a compression ratio of 144.0, which means that the compressed data is 144 times smaller than the original data. The patch sizes are chosen to be one day, half a day, and a third of a day respectively. This results in slightly larger compression ratios than the compression ratios of the Kronecker decomposition. We will see in Table 3.3 and Figure 3.6 that this does not significantly skew the results. The RMSE of the Kronecker decomposition is significantly smaller than the RMSE of the averaging method for all ranks.

Kronecker decomposition			Averaging			
Rank	RMSE (°C)	CR	RMSE Rep. (°C)	RMSE Int. (°C)	CR Rep.	CR Int.
1	2.06	142.3	2.38	2.33	66.0	66.0
2	1.19	71.2	1.44	1.48	33.0	44.0
3	0.89	47.5	0.90	1.07	18.0	33.0

Table 3.3: Comparison of the compression ratio (CR) of Kronecker decomposition and two averaging methods. For the comparison we selected for each rank of the Kronecker decomposition a patch size for the averaging methods that results in a root mean square error (RMSE) in °C that is larger or equal to the RMSE of the Kronecker decomposition.

Table 3.3 shows the compression ratio of the Kronecker decomposition compared to the compression ratio of the averaging and the interpolation method respectively. For this comparison, we selected for each rank of the Kronecker decomposition a patch size for the averaging and the interpolation method that results in an RMSE that is larger or equal to the RMSE of the Kronecker decomposition. Note that the RMSE of the averaging and the interpolation method is in most cases significantly larger than the RMSE of the Kronecker decomposition. We still achieve a compression ratio that is more than two times larger for the Kronecker decomposition compared to the simple averaging method and more than 40% larger compared to the interpolation method in the worst case.

Expanding this to higher ranks, we can plot the RMSE against the compression ratio for the Kronecker decomposition and the averaging methods, as shown in Figure 3.6.

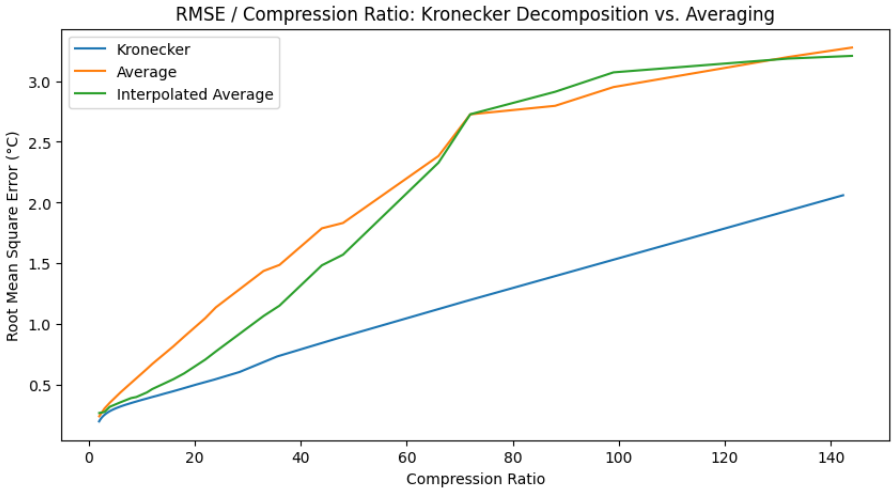


Figure 3.6: Approximation root mean square error (RMSE) vs. compression ratio for the Kronecker decomposition and the averaging methods. The Kronecker decomposition is shown for ranks 1 to 72, which corresponds to a compression ratio between 2 and 143. The averaging methods are shown for patch sizes between 2 and 150.

As will be discussed in Chapter 5, it is possible to evaluate queries over the Kronecker decomposition without reconstructing the data. The same is true for the simple averaging method, but not for the averaging with linear interpolation method, where it would be necessary to reconstruct the data first. It therefore is an unfair comparison for the Kronecker decomposition. However, we still want to compare the Kronecker decomposition to the interpolation method, because we only compare the reconstruction accuracy.

We conclude this section by stating that the Kronecker decomposition can capture the daily period and its variations in the temperature data exceptionally well. Compared to the averaging

and the interpolation methods, the Kronecker decomposition achieves a better compression ratio and a better reconstruction accuracy. The Kronecker decomposition is therefore a good choice at least for this specific dataset.

Chapter 4

Data Representation in KroneDB

In this chapter, we will look at the application of the Kronecker decomposition in KroneDB. For this purpose, in Section 4.1, it is first explained how a KroneRelation is defined and how to decompose a KroneRelation using the Kronecker decomposition. The following sections discuss different choices the user has to make when decomposing a KroneRelation. The impacts of these choices on the approximation result are immediately shown and explained on the specific dataset used in the motivating example (Section 3). Section 4.2 discusses three different ways to handle the key columns, which contain potential join keys. Section 4.3 discusses the choice of the rank and the period length, which influence the compression ratio and the approximation accuracy. The effects of negative values in the data and how a shift of the whole dataset influences the approximation are discussed in Section 4.4. Similar to the choice of the key decomposition in Section 4.2, Section 4.5 discusses the choice of the column decomposition if multiple time series are stored in the same relation and are decomposed together. The final section, Section 4.6, shows how the observations made generalize to other time series on three examples.

The main takeaways of this chapter are that only the data columns of a KroneRelation are decomposed using the Kronecker decomposition. The simplest and always possible choice to handle the key columns is to do an independent decomposition for each unique combination of key values, which will be used in the rest of this thesis. A higher rank generally leads to a better approximation but can become computationally expensive later on. The period length should be chosen to be a multiple of the natural period in the data. Negative values have the effect of mirroring the period around the origin, which can be a problem for the approximation. Shifting the data by a constant value can be used to avoid negative values, but has an additional effect on how the period amplitude is scaled, which can impact the approximation in a major way. The collective column decomposition can use the mutual information between similar time series to improve the compression ratio, compared the individual column decomposition. Finally, the observations made in this chapter generalize well to the other time series datasets that are tested.

4.1 KroneRelation

A KroneRelation is a relation with a clear separation between the *key columns*, and the *value columns*. Only the *key columns* can be used as join keys to join with other relations. The *value columns* are the columns containing the dense numerical values that we want to decompose using the Kronecker decomposition. They can not be used as join keys. This has the advantage that the *value columns* can be decomposed without affecting the join keys and allows for additional structural compression techniques, such as factorized computation [OS16]. Because the Kronecker decomposition relies on the ordering of the rows in the *data-matrix*, we need to make sure that the tuples in the KroneRelation have a specific order. To define this order, each KroneRelation has an *index column* that contains an orderable value for each tuple. This *index column* together with the *key columns* must create a unique identifier for each tuple in the KroneRelation.

The original KroneRelation which is put into KronDB is called the *data-relation* and is denoted as

$$\mathbf{D}(\mathbf{x}, rid, \mathbf{d}),$$

where $\mathbf{x} = x_1, \ldots, x_n$ are the key columns, rid (row index) is the index column and $\mathbf{d} = d_1, \ldots, d_{n_d}$ are the value columns. The number of key columns is denoted as n and the number of value columns, which is the width of the data-matrix, is denoted as n_d.

The data-matrix $D \in \mathbb{R}^{m_d \times n_d}$ is the matrix containing the value columns of the data-relation, also called the *data-columns*. The number of tuples in the data-relation is equal to the height of the data-matrix m_d. The rows of the data-matrix are ordered by the index column. In the general case, each unique combination of the key values creates a new data-matrix. Thus, the data-relation $\mathbf{D}(x_1, x_2, rid, d_1, d_2, d_3, d_4)$ defined as

x_1	x_2	rid	d_1	d_2	d_3	d_4
a	b	1	d_{11}	d_{12}	d_{13}	d_{14}
a	b	2	d_{21}	d_{22}	d_{23}	d_{24}
a	b	3	d_{31}	d_{32}	d_{33}	d_{34}
a	b	4	d_{41}	d_{42}	d_{43}	d_{44}
a	c	1	d_{51}	d_{52}	d_{53}	d_{54}
a	c	2	d_{61}	d_{62}	d_{63}	d_{64}
a	c	3	d_{71}	d_{72}	d_{73}	d_{74}
a	c	4	d_{81}	d_{82}	d_{83}	d_{84}

contains two data matrices

$$D_{ab} = \begin{bmatrix} d_{11} & d_{12} & d_{13} & d_{14} \\ d_{21} & d_{22} & d_{23} & d_{24} \\ d_{31} & d_{32} & d_{33} & d_{34} \\ d_{41} & d_{42} & d_{43} & d_{44} \end{bmatrix} \text{ and } D_{ac} = \begin{bmatrix} d_{51} & d_{52} & d_{53} & d_{54} \\ d_{61} & d_{62} & d_{63} & d_{64} \\ d_{71} & d_{72} & d_{73} & d_{74} \\ d_{81} & d_{82} & d_{83} & d_{84} \end{bmatrix}.$$

4.1.1 Decomposed KroneRelations

As seen in Section 2.1.3, the Kronecker decomposition decomposes the data-matrix $D \in \mathbb{R}^{m_d \times n_d}$ into the scaling matrices $S^r \in \mathbb{R}^{m_s \times n_s}$ and period matrices $P^r \in \mathbb{R}^{m_p \times n_p}$ for each rank r.

After decomposing a KroneRelation, we get two new KroneRelations, one for the scaling-matrices and one for the period-matrices which are called *scaling-relation* and *period-relation* respectively.

The decomposed relations need additional information about how they handle the ranks of the decomposition. For this purpose, we will add a *rank column* to the scaling- and the period-relation. The scaling- and period-relation are then denoted as

$$\mathbf{S}(\mathbf{x}, rid_s, \mathbf{s}, r) \text{ and } \mathbf{P}(\mathbf{x}, rid_p, \mathbf{p}, r),$$

where $\mathbf{x} = x_1, \ldots, x_n$ are the key columns, rid_s and rid_p are the index columns, $\mathbf{s} = s_1, \ldots, s_{n_s}$ and $\mathbf{p} = p_1, \ldots, p_{n_p}$ are the value columns and r is the rank column. n_s is the number of value columns in the scaling-relation and n_p is the number of value columns in the period-relation, which are also the widths of the scaling-matrix and the period-matrix respectively, therefore $n_d = n_s \cdot n_p$.

If the data-relation has one data-matrix for each unique combination of the key values, then the scaling and period-relations will have a scaling and a period-matrix for each unique combination of the key values, because every data-matrix will be decomposed individually. Decomposing the data-relation $\mathbf{D}(x_1, x_2, rid, d_1, d_2, d_3, d_4)$ from the previous section means decomposing each data-matrix D_{ab} and D_{ac} separately. Using a rank-2 decomposition and a 2×2 period-matrix, we get

the following scaling and period matrices:

$$D_{ab} \approx \begin{bmatrix} s_{11} & s_{12} \\ s_{21} & s_{22} \end{bmatrix} \otimes \begin{bmatrix} p_{11} & p_{12} \\ p_{21} & p_{22} \end{bmatrix} + \begin{bmatrix} s_{31} & s_{32} \\ s_{41} & s_{42} \end{bmatrix} \otimes \begin{bmatrix} p_{31} & p_{32} \\ p_{41} & p_{42} \end{bmatrix} = S^1_{ab} \otimes P^1_{ab} + S^2_{ab} \otimes P^2_{ab}$$

and

$$D_{ac} \approx \begin{bmatrix} s_{51} & s_{52} \\ s_{61} & s_{62} \end{bmatrix} \otimes \begin{bmatrix} p_{51} & p_{52} \\ p_{61} & p_{62} \end{bmatrix} + \begin{bmatrix} s_{71} & s_{72} \\ s_{81} & s_{82} \end{bmatrix} \otimes \begin{bmatrix} p_{71} & p_{72} \\ p_{81} & p_{82} \end{bmatrix} = S^1_{ac} \otimes P^1_{ac} + S^2_{ac} \otimes P^2_{ac}.$$

These scaling and period matrices are then put into the scaling and period-relations together with the corresponding key values to get the decomposed KroneRelations

$$\mathbf{S}(x_1, x_2, rid_s, s_1, s_2, r) \text{ and } \mathbf{P}(x_1, x_2, rid_p, p_1, p_2, r)$$

containing the tuples

x_1	x_2	rid_s	s_1	s_2	r
a	b	1	s_{11}	s_{12}	1
a	b	2	s_{21}	s_{22}	1
a	b	1	s_{31}	s_{32}	2
a	b	2	s_{41}	s_{42}	2
a	c	1	s_{51}	s_{52}	1
a	c	2	s_{61}	s_{62}	1
a	c	1	s_{71}	s_{72}	2
a	c	2	s_{81}	s_{82}	2

and

x_1	x_2	rid_p	p_1	p_2	r
a	b	1	p_{11}	p_{12}	1
a	b	2	p_{21}	p_{22}	1
a	b	1	p_{31}	p_{32}	2
a	b	2	p_{41}	p_{42}	2
a	c	1	p_{51}	p_{52}	1
a	c	2	p_{61}	p_{62}	1
a	c	1	p_{71}	p_{72}	2
a	c	2	p_{81}	p_{82}	2

respectively. In the definition of a KroneRelation, the rank column is one of the key columns and it would be possible to decompose **S** and **P** further.

The index columns rid_s and rid_p map the tuples in the scaling- and period-relation to the tuples in the data-relation according to (2.2). This means that the values $d_1, \ldots, d_{n_d}$ in a tuple in the data-relation with the keys $x_1, \ldots, x_n$ and index rid are approximated by the values $s_1, \ldots, s_{n_s}$ in the tuples in the scaling-relation with the keys $x_1, \ldots, x_n$ and index rid_s and the values $p_1, \ldots, p_{n_p}$ in the tuples in the period-relation with the keys $x_1, \ldots, x_n$ and index rid_p, such that

$$rid = (rid_s - 1) \cdot m_p + rid_p. \tag{4.1}$$

The value columns are mapped accordingly such that

$$d_j \text{ is decomposed into } s_{j_s} \text{ and } p_{j_p} \text{ iff } j = (j_s - 1) \cdot n_p + j_p. \tag{4.2}$$

Consider the value d_{11} in the data-relation. It is the value for the column d_1 in the first tuple together with the index $rid = 1$. Using (4.1), we get that $rid_s = 1$ and $rid_p = 1$ and using (4.2), we get that $j_s = 1$ and $j_p = 1$. The approximation of d_{11} is therefore calculated as

$$d_{11} \approx s_{11} \cdot p_{11} + s_{31} \cdot p_{31}.$$

4.2 Individual vs. Collective Key Decomposition

In this section, alternative ways to decompose a data-relation into a scaling-relation and a period-relation are discussed. This avoids having a separate Kronecker decomposition for each unique combination of key values. To make the examples clearer, we will only use a single key column, which is conceptually equal to having multiple key columns and treating each unique combination of key values separately. Additionally, for simplicity, a rank-1 decomposition is assumed and the rank column is omitted.

Note that all further sections in the thesis always assume the individual key decomposition. This includes all experiments. It is trivial to extend all theoretical results to the collective key decomposition, but it is not the focus of this thesis.

4.2.1 Individual Key Decomposition

The first example is the individual key decomposition, which has already been used in Section 4.1.1. For each key, there is a separate decomposition and thus, a separate scaling-matrix and period-matrix. To decompose the data-relation, we want to group by the key columns and set the *rid* column for each group. A data relation together with its decomposed relations could look like this:

Data-Relation

key	rid	value
a	1	d_1
a	2	d_2
a	3	d_3
b	1	d_4
b	2	d_5
b	3	d_6
b	4	d_7

Scaling-Relation

key	rid	value
a	1	s_1
b	1	s_2
b	2	s_3

Period-Relation

key	rid	value
a	1	p_1
a	2	p_2
a	3	p_3
b	1	p_4
b	2	p_5

.

4.2.2 Collective Key Decomposition

The first variation is the case where we have a separate scaling-matrix for each unique key value, but the period-matrix is shared between all key values and the second variation is the other way around. Both variations are only possible under certain conditions. The influence on the approximation accuracy and compression ratio of sharing the period-matrix or the scaling-matrix is comparable to the results in Section 4.5.2 where the period-matrix and scaling-matrix are shared between columns.

Shared Period Consider a decomposition, where the period length is the same for every key. Thus, the period-matrix can be shared between all key values. The data-relation must be ordered to build blocks of m_p consecutive rows with the same key value, where m_p is the common period length. The *rid* column is shared between all key values. The key column can be removed from the period-relation and the decomposition can be done as follows:

Data-Relation

key	rid	value
a	1	d_1
a	2	d_2
b	3	d_3
b	4	d_4
b	5	d_5
b	6	d_6

Scaling-Relation

key	rid	value
a	1	s_1
b	2	s_2
b	3	s_3

Period-Relation

rid	value
1	p_1
2	p_2

.

Shared Scaling If we have the same number of periods for each key, the scaling-matrix can be shared. In this case, we can stack the periods for each key value into a single matrix. The data-relation is ordered such that each set of consecutive rows with the same key value has a length that is exactly the period length for this key value. This means that we get an alternating pattern of key values in the data-relation that is repeated after each period length, where the period length is the length of the combined period that is shared between all key values, which in

this example is $m_p = 3$. The key column can be removed from the scaling-relation. Thus,

Data-Relation

key	rid	value
a	1	d_1
b	2	d_2
b	3	d_3
a	4	d_4
b	5	d_5
b	6	d_6

Scaling-Relation

rid	value
1	s_1
2	s_2

Period-Relation

key	rid	value
a	1	p_1
b	2	p_2
b	3	p_3

.

4.3 Rank and Period Length

Looking back at the motivating example in Section 3, we started with the data at hand and concluded that the period length should correspond to a single day to exploit the structure of the daily period and get a good approximation of the data. There are other considerations to be taken into account when choosing the period length, even if the data has an obvious and regular period.

One aspect is the compression ratio (CR). The compression ratio is calculated by

$$CR = \frac{m_d \cdot n_d}{(m_s \cdot n_s + m_p \cdot n_p) \cdot k}$$

where $m_d \times n_d$ is the size of the data matrix, $m_s \times n_s$ is the size of the scaling-matrix, $m_p \times n_p$ is the size of the period-matrix and k is the rank of the Kronecker decomposition. For this section, it is assumed that the number of value columns n_d, n_s, and n_p is fixed. How the number of value columns in the scaling- and period-matrix can be varied is discussed in Section 4.5. Therefore, the compression ratio is defined as

$$CR = \frac{m_d}{(m_s + m_p) \cdot k} \ .$$

To get a large rank for a given compression ratio, a period length that is as close as possible to the square root of the number of data points must be chosen, because $m_d = m_s \cdot m_p$. If we want a small rank, we need to choose a very small (or very large) period length to keep the compression ratio constant. We will see in this section that if we want a low rank, we should always choose a small period length, instead of a large one.

Rank Having a higher rank for the same compression ratio means that more levels of detail in the data can be captured. We can think about this as encoding structural information that has a certain frequency in each rank. The more ranks there are, the more frequencies are persisted in the Kronecker decomposition. They are not actual frequencies, as would be the case in a Discrete Fourier Transform (DFT) based compression, but it is structural information that applies to a certain subset of the data. Consider the Kronecker decomposition with a period length of 1.5 days in Figure 4.5, where the first three ranks encode something that looks like different frequencies.

In general, to get a better approximation of the original data, a higher rank allows us to capture more levels of detail in the data. However, we need to be aware that certain aggregates are more expensive to calculate for higher ranks. We will go into specifics about the problem of calculating aggregates over high-rank Kronecker decompositions in Section 5.3.7.

Period Length It is generally preferable to keep the period relatively small because the period is completely static (per rank) and can only be scaled by a scaling factor. This means that trends that have a smaller frequency than the period but differ from one period to the next cannot be represented in a single rank.

Consider the temperature measurements from the motivating example in Section 3. The assumption was that the temperature has a daily period and that the period length should therefore be one day. However, the temperature also has a natural yearly period and the period length could also be one year. Instead of 12,000 days where each day can be scaled independently, we would have 33 years where all days in a year are scaled together. The comparison of the two Kronecker decompositions is shown in Figure 4.1. The period matrices of the two Kronecker decompositions look almost as if they were describing the same structure. However, the period in the first row describes the structure of a single day, while the period in the second row describes the structure of a whole year. The yearly period fluctuates a lot because it tries to capture the structure of all 365 days in a single function. The daily period on the other hand is very smooth, because it only needs to capture the structure of a single day, and the structure of the years is captured by the scaling-matrix. Looking at the approximation, we can see that the Kronecker decomposition with the daily period can capture the variation in the data much better than the Kronecker decomposition with the yearly period, even though the compression ratio is much higher for the daily period. The approximation of the yearly Kronecker decomposition looks like an average of the data, what it essentially is, because it is forced to have a uniform year, which will be a type of average of all the years in the data.

The comparison of the two decompositions for further ranks is shown in Figure 4.2. As we can see, choosing a yearly period is not a good idea, because it is significantly worse at capturing the structure of the data for low and for high ranks.

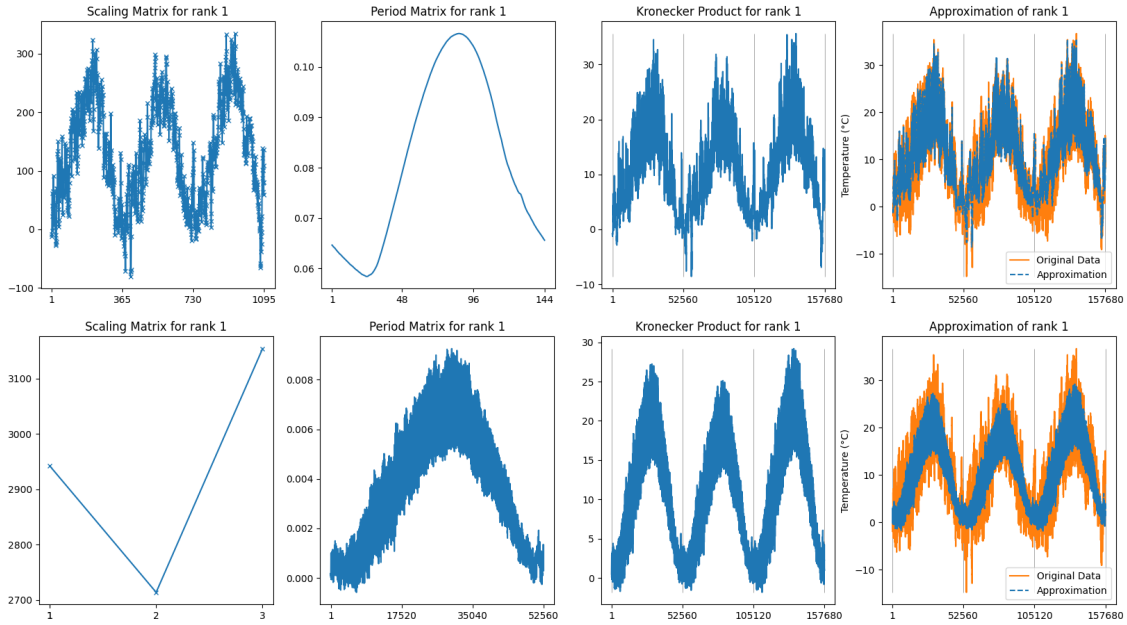


Figure 4.1: Comparison of the Kronecker decompositions with period length 1 day and 1 year for the last 3 years of the temperature dataset. The first row shows the Kronecker decomposition with a period length of 1 day, where the period-matrix describes the structure of a single day. The second row shows the Kronecker decomposition with a period length of 1 year, where the period-matrix describes the structure of a whole year.

For every period length, there is a mirrored period length, which is the number of scaling factors. This means for every period length m_p, where $m_p \cdot m_s = m_d$, there is a mirrored period length m'_p, where $m'_p \cdot m'_s = m_d$, such that $m'_p = m_s$ and $m'_s = m_p$. A selection of mirrored periods is plotted in Figure 4.3. The main observation to be made is that it is usually better to choose a period length that is smaller than the square root of the number of data points, i.e. to

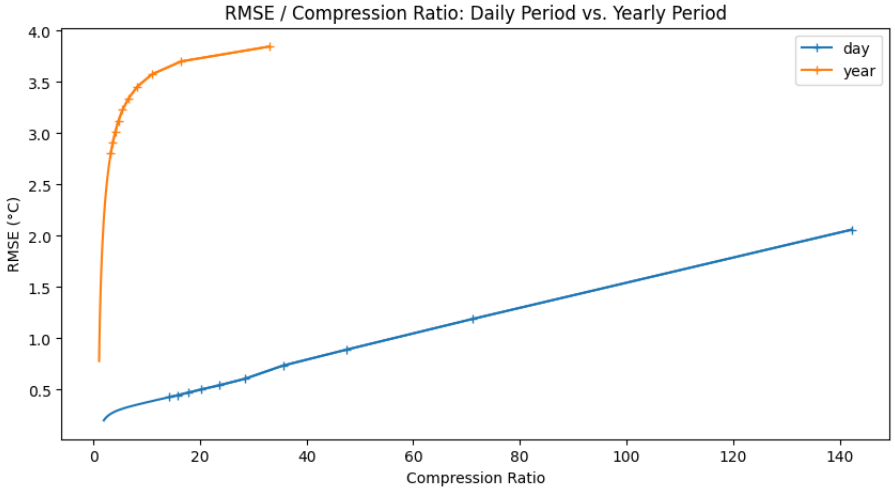


Figure 4.2: Comparison of the approximation error for daily and yearly periods for different ranks. The error is calculated as the root mean squared error (RMSE) between the original data and the approximation. The error for the daily period is significantly lower than the error for the yearly period. The first ten ranks for each period length are marked using a "+" sign, where rank 1 is the rightmost rank with the highest compression ratio and rank 10 is the leftmost rank with the lowest compression ratio.

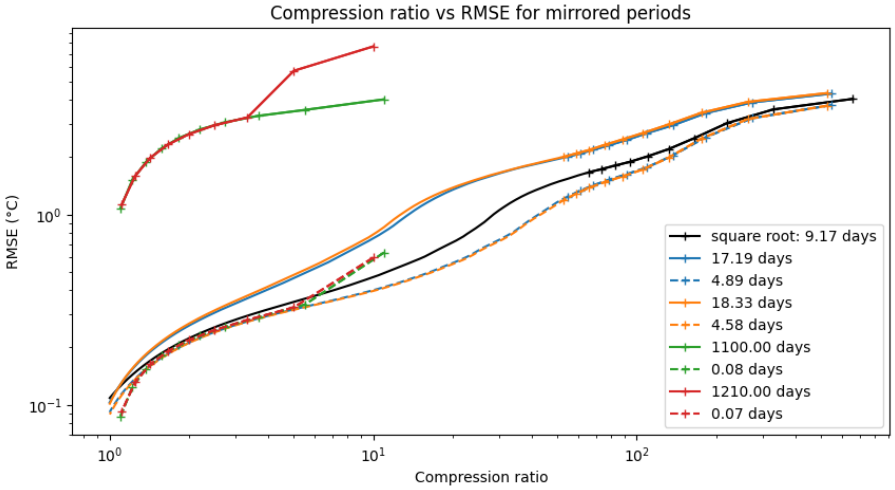


Figure 4.3: Mirrored Periods for different period lengths. The mirrored period pairs are plotted in the same color. The smaller periods, with a period length smaller than the square root, are dashed. The square root in black is the mirrored period of itself. The first ten ranks for each period length are marked using a "+" sign, where rank 1 is the rightmost rank with the highest compression ratio and rank 10 is the leftmost rank with the lowest compression ratio.

choose m_p and m_s such that $m_p \leq m_s$. Note that the smallest period lengths perform worse than the larger period lengths for rank 1. However, from rank 2 on, the smaller period lengths perform similar or better than the larger period lengths and thus should be preferred.

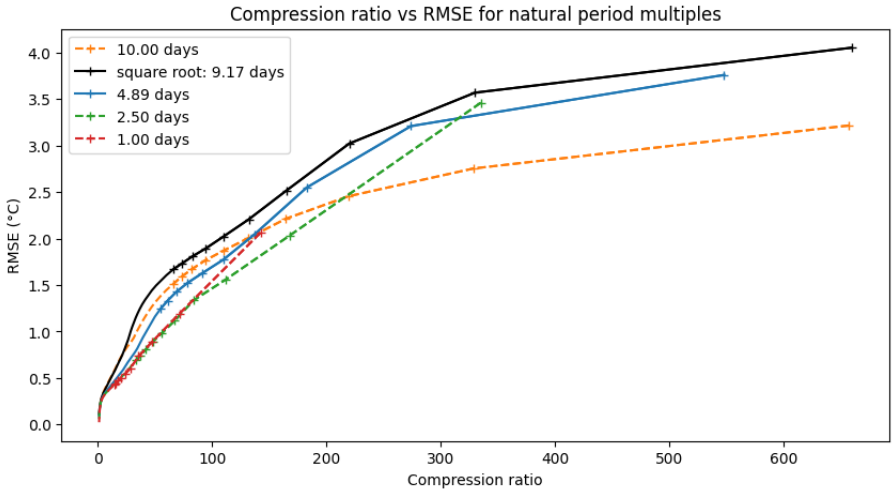


Figure 4.4: Different multiples of the natural period. The first ten ranks for each period length are again marked using a "+" sign.

We already assumed that the natural period should perform best. In Figure 4.4, it can seen that not only the natural period performs exceptionally well, but also multiples of the natural period. Even periods that are larger than the square root of the number of data points perform well if they are multiples of the natural period, at least for small ranks. It is also important to note that not only multiples of the natural period perform exceptionally well, but also multiples of the natural period plus half the natural period, but only for ranks higher than 1. They only become good for rank 2 and higher because their period needs to fit two different cases, where the first case is a period that starts with a full day and ends with a half day, and the second case is a period that starts with a half day and ends with a full day.

This phenomenon is shown in Figure 4.5. Looking at rank 1, we can see that the daily structure is gone, because the period is not a multiple of the natural period. The period-matrix looks like it tries to capture the structure of the two overlapping 1.5-day periods. The approximation compared to the real data shows that it is not able to capture much and it almost looks like a constant representing the average temperature of the 1.5 days, which is encoded into the scaling-matrix. However, in rank 2, the period-matrix can correct the structure of the two overlapping 1.5-day periods quite well by having the structure almost centered around the origin and thus being able to flip it around by using negative scaling factors for every second period. Comparing the approximations of rank 2 and rank 3 shows another problem of the longer period. The second period, in between samples 216 and 432, starts with a half-day and ends with a full day. The third rank now adjusts this full day by stretching it vertically to fit the real day much better, but at the same time, it compresses the prepended half-day, which seems to make this half-day worse than it was in the rank 2 approximation.

We conclude that the best period length is the natural period or a small multiple of it. If this does not work, we can try a multiple of half the natural period, but we should be aware that this will only work well for ranks higher than 1. In general, if we do not have a natural period, we should try to find a period that is smaller than the number of scaling factors, which means smaller than the square root of the number of data points.

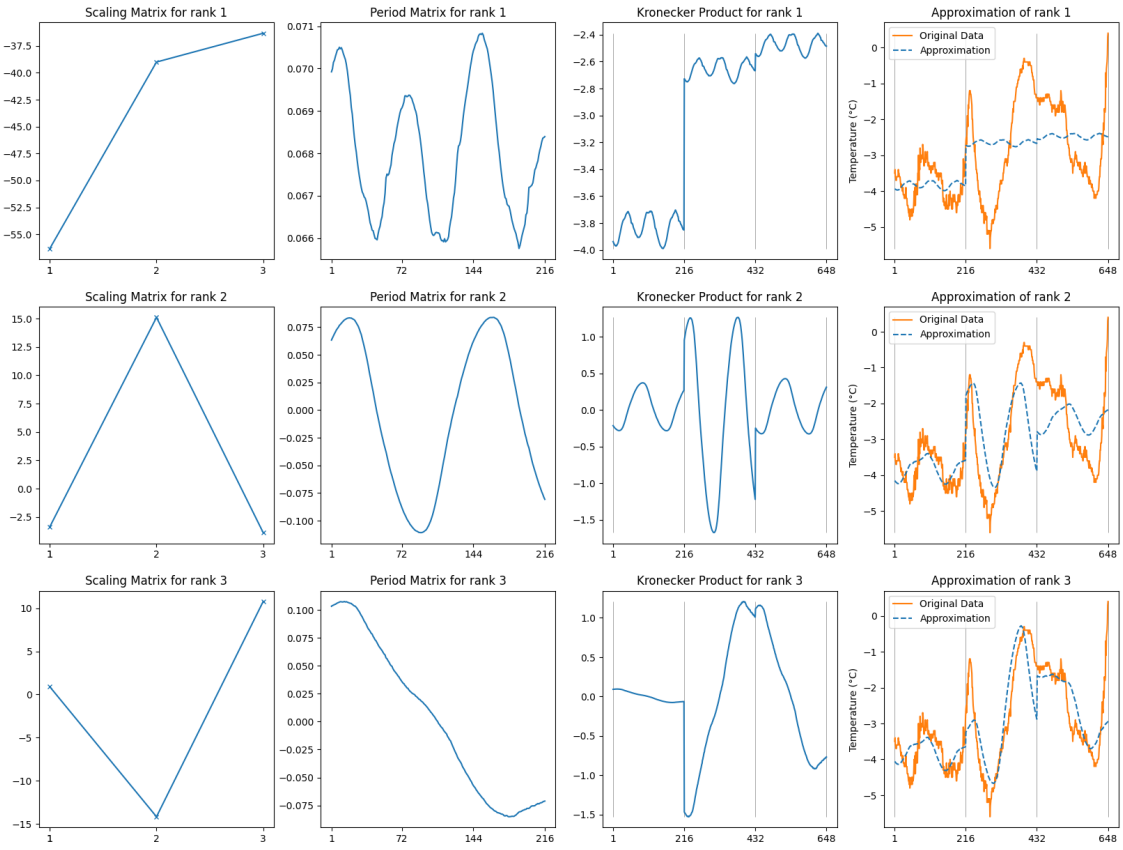


Figure 4.5: Kronecker decomposition for the first 4.5 days of our temperature dataset decomposed with a period of 1.5 days. The same days decomposed with a single day period is shown in Figure 4.6.

4.4 Negative Values and Amplitude Scaling

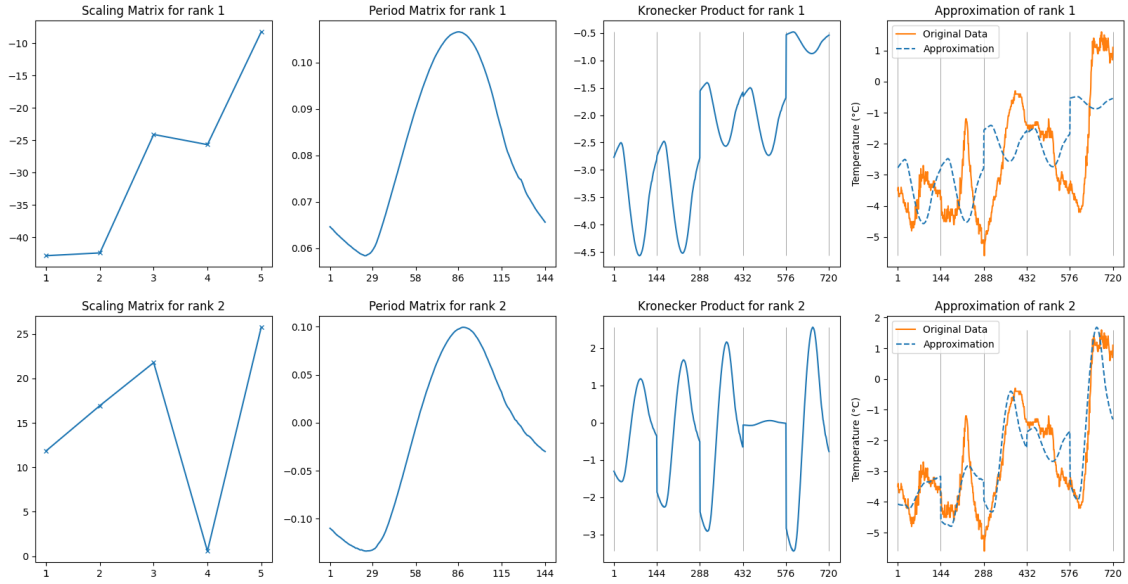


Figure 4.6: Kronecker decomposition of the first five days of the temperature data in Kloten ZH. The data is in winter, so the temperature is mostly below 0°C.

We have shown how the Kronecker decomposition encodes the data by showing a window of the last five days, which are in summer, in Figure 3.5. We already mentioned, that there might be a problem if the data has negative values. The Kronecker decomposition for the first five days, which are in winter and have mostly temperatures below 0°C, is shown in Figure 4.6. We can see that the Kronecker decomposition is not able to maintain the structure of the daily period in the first rank. The reason is, that to achieve negative values, we need to have negative scaling factors. A negative scaling factor comes with the side effect, that the period is inverted because every value of the period is multiplied by a negative number. This is why the peaks in the original data, are valleys in the rank-1 approximation and vice versa.

Looking at rank 2 we can see that this rank stretches the period with high positive scaling factors. This is how the second rank can compensate for the inverted period of the first rank and force the period back in the right direction. We can also see that the second rank does almost nothing on the 4th day because it does not follow the typical pattern of the other days. This is why the second rank has a very low scaling factor for the 4th day.

In general, these five winter days have a very different structure than the summer days. Not only do they introduce negative values, but they also have a much smaller range of values. We will first address the problem of negative values, and then look at what the smaller value range means for the Kronecker decomposition.

Negative Values To avoid the inverted period, we need to get rid of the negative values. This can be done by shifting the data into the positive domain. This shift must be reversed to reconstruct or query the data. How this shift can be recovered while querying the data is discussed in Section 5.3.6.

Consider the Kronecker decomposition of the first five days of the temperature data in Kloten ZH, after shifting it into the positive domain, as shown in Figure 4.7. The shifting is done by subtracting the minimum value of the data, which is negative, from all values. We can see that the Kronecker decomposition is now able to maintain the structure of the daily period in the first

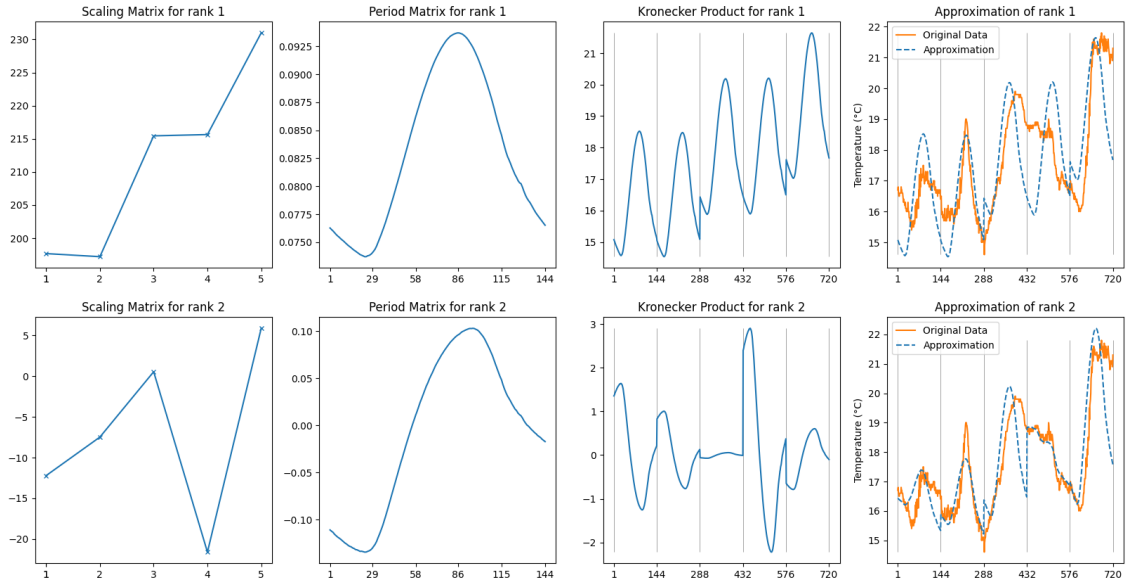


Figure 4.7: Kronecker decomposition of the first five days of the temperature data in Kloten ZH after shifting it into the positive domain.

rank, which should improve the accuracy of the approximation. However, we can also see that for the first day, we still do not have a very good approximation. This is because the first day has a very narrow range of values, which cannot be represented well by the first rank. In order to compress the period the scaling factor needs to be much smaller. This would also make the average temperature of the first day much lower than it is, which is why the scaling factor is not that small. We will look further into this in the next paragraph. The most important aspect is the effect of the shift on the error of the approximation. To see the difference between the shifted and the unshifted Kronecker decomposition, we plotted the RMSE over the compression ratio curve for both in Figure 4.8.

The error of the shifted Kronecker decomposition is not much smaller than the error of the unshifted Kronecker decomposition. The main reason for this mediocre improvement is what was discussed before. The much larger amplitude for days that have a small range hurts the accuracy of the approximation. This effect diminishes the improvement we get from the shift. Overall the whole dataset we gained about 3.0% in error for the first rank, where we had an error of 2.061°C for the original dataset and dropped it to 1.997°C for the shifted dataset. For the second rank, we only get a 0.6% smaller error, and for the third rank, we even have an error that is 1.0% larger. All of the other ranks have an error that is not significantly different from the error of the unshifted Kronecker decomposition. It is important to note that this effect is highly dependent on the data.

Period amplitude scaling In this paragraph, we discuss the effect that almost mitigated the improvement we got from shifting the data into the positive domain, which is the scaling of the period amplitude. It is obvious that scaling (or multiplying) a periodic function like a sine or cosine with a constant will change the amplitude of the function. This is exactly what happens with our period. But in our case, the scaling is primarily used to shift the period up and down to fit it to the average of the period in the data.

These two effects cannot be separated, but we can adjust the magnitude of the amplitude scaling by shifting the data closer to the origin, to make it more prominent, or further away from the origin, to make it less prominent. By shifting the data away from the origin, we decrease the relative difference between the data points while keeping the absolute difference. This can

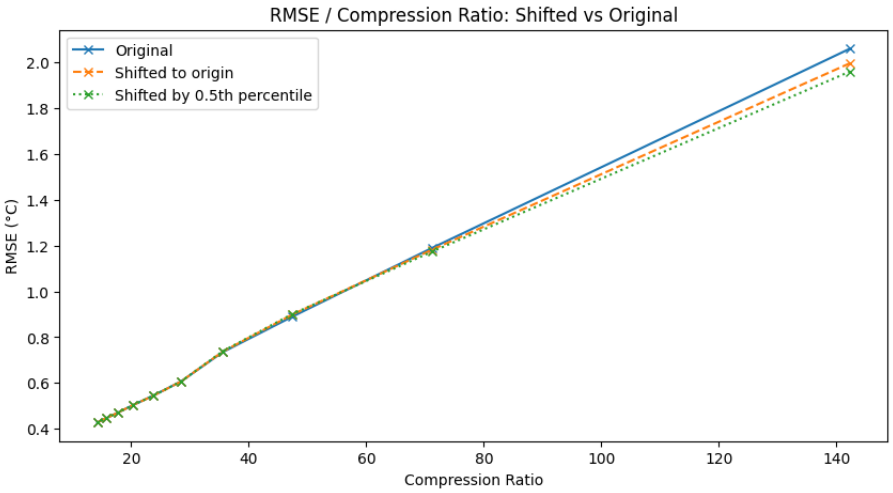


Figure 4.8: Comparing the error of the Kronecker decomposition with and without shifting the data into the positive domain. The error is calculated as the mean squared error between the original data and the approximation. The ranks one to ten are marked with an x.

mitigate the effect of the amplitude scaling if we shift the data far enough away from the origin.

Our first intuition to handle the negative values for the temperature data was to switch to Kelvin, which would make all values positive. The problem was that all the values would be relatively large, between 250K and 320K, which resulted in a worse compression than the original data, because of this effect. Warm days have a much larger amplitude than cold days, but by shifting the data away from the origin, we force the amplitude to be the very similar for all days.

In the other direction, we cannot shift the data closer to the origin than the minimum value of the data, because we would get negative values again. In some cases, this might be a good idea, even if the period gets inverted because the effect of the period amplitude might dominate the effect of the flipped period. This can be seen in our temperature example, where for a rank 3 approximation, this is the case. We can see this effect if we look at the summer days in Figure 4.9 and compare it to the summer days in the original dataset in Figure 3.5.

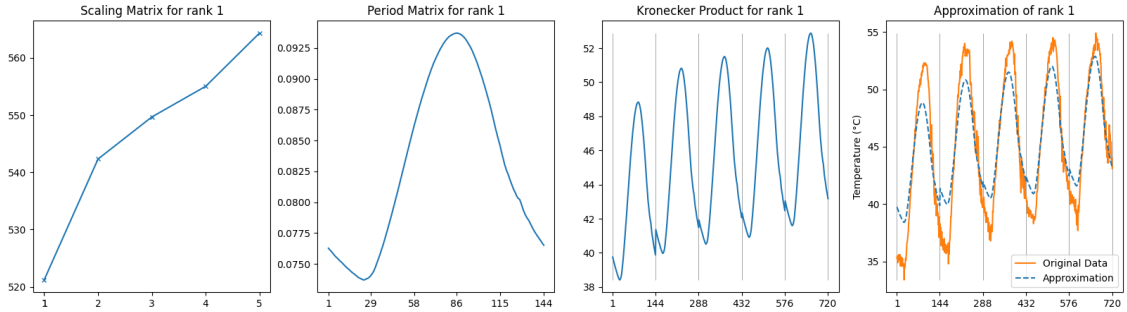


Figure 4.9: Kronecker decomposition of the last five days of the temperature data in Kloten ZH after shifting it into the positive domain.

The rank 1 approximation of these summer days is much worse than the approximation before the shift because the period amplitude is not large enough. And it is not large enough, because the low-temperature measurements that need a small amplitude are now much further away from the origin and therefore are more influenced by the scaling. That means that overall the period amplitude is smaller to balance the too-large amplitude for the winter days and the too-small

amplitude for the summer days.

Considering this tradeoff, we can see that the shift into the positive domain is not always a good idea. That a rank 3 approximation can be better in the original domain, even though ranks 1 and 2 are worse, as seen in the previous paragraph, suggests that it might be easier for higher ranks to compensate for the inverted period, than to compensate for the too large or too small period amplitude. We can try to find a sweet spot for the shift, where the number of negative values is minimized, but we still keep the periods with a small amplitude close to the origin. In our case, we tested it by shifting the data by the 0.5th percentile of the data, which is -8.1°C compared to the minimum of -20.2°C and got marginal but not significant improvements, which is also shown in Figure 4.8.

4.5 Individual vs. Collective Column Decomposition

In this section, we will discuss how to decompose multiple time series together. In the Sections 3, 4.3 and 4.4 we only looked at decomposing a single time series. In general, the decomposition can be done over multiple time series, which results in a data-relation with multiple value columns, as seen conceptually in Section 4.1.

4.5.1 Individual Column Decomposition

The simplest way to decompose multiple columns is to decompose each column individually. The Kronecker decomposition is applied to each column separately, which means that each column has a data-matrix and therefore also a scaling-matrix and period-matrix, which are all single-column matrices. The scaling- and period-matrices are then concatenated to form the overall scaling- and period-matrices respectively. To express this case, $\circledast$ is used as the column-wise Kronecker product to get

$$
\begin{aligned}
D &= \begin{bmatrix} s_{11}\mathbf{p}_1 & s_{12}\mathbf{p}_2 & \dots & s_{1n_d}\mathbf{p}_{n_d} \\ s_{21}\mathbf{p}_1 & s_{22}\mathbf{p}_2 & \dots & s_{2n_d}\mathbf{p}_{n_d} \\ \vdots & \vdots & \ddots & \vdots \\ s_{m_s1}\mathbf{p}_1 & s_{m_s2}\mathbf{p}_2 & \dots & s_{m_sn_d}\mathbf{p}_{n_d} \end{bmatrix} \\
&= \begin{bmatrix} s_{11} & s_{12} & \dots & s_{1n_d} \\ s_{21} & s_{22} & \dots & s_{2n_d} \\ \vdots & \vdots & \ddots & \vdots \\ s_{m_s1} & s_{m_s2} & \dots & s_{m_sn_d} \end{bmatrix} \circledast \begin{bmatrix} p_{11} & p_{12} & \dots & p_{1n_d} \\ p_{21} & p_{22} & \dots & p_{2n_d} \\ \vdots & \vdots & \ddots & \vdots \\ p_{m_p1} & p_{m_p2} & \dots & p_{m_pn_d} \end{bmatrix} = S \circledast P,
\end{aligned}
$$

where $n_d = n_s = n_p$ and $\mathbf{p}_i$ is the i-th column of the period-matrix.

The column mapping of the KroneRelations can therefore be simplified from (4.2) to

$$
d_j \text{ is decomposed into } s_{j_s} \text{ and } p_{j_p} \text{ iff } j = j_s = j_p \tag{4.3}
$$

if the column-wise Kronecker product, i.e. the individual column decomposition, is used.

4.5.2 Collective Column Decomposition

A collective column decomposition means that the data-matrix is decomposed as a whole using the Kronecker decomposition. Thus,

$$
\begin{aligned}
D &= \begin{bmatrix} s_{11}P & s_{12}P & \dots & s_{1n_s}P \\ s_{21}P & s_{22}P & \dots & s_{2n_s}P \\ \vdots & \vdots & \ddots & \vdots \\ s_{m_s1}P & s_{m_s2}P & \dots & s_{m_sn_s}P \end{bmatrix} \\
&= \begin{bmatrix} s_{11} & s_{12} & \dots & s_{1n_s} \\ s_{21} & s_{22} & \dots & s_{2n_s} \\ \vdots & \vdots & \ddots & \vdots \\ s_{m_s1} & s_{m_s2} & \dots & s_{m_sn_s} \end{bmatrix} \otimes \begin{bmatrix} p_{11} & p_{12} & \dots & p_{1n_p} \\ p_{21} & p_{22} & \dots & p_{2n_p} \\ \vdots & \vdots & \ddots & \vdots \\ p_{m_p1} & p_{m_p2} & \dots & p_{m_pn_p} \end{bmatrix} = S \otimes P,
\end{aligned}
$$

where $n_s \cdot n_p = n_d$, which is the usual Kronecker decomposition as described in Section 2.1.

In this section, we are looking at an example of multiple time series that share the same range of values. We will look at the MeteoSwiss temperature data from other cities in Switzerland and decompose them together with the temperature data from Kloten ZH, which we already saw in the previous sections. For this collective decomposition, we have multiple options on what part of the data we want to decompose together. As discussed in Section 2.1, we can choose the width of the scaling- and period-matrix. The extreme cases are if either of the two decomposed matrices is a single column. In other words, if the time series share the same scaling or the same period. Note that the choice between these two extremes is the same as the choice between shared scaling and shared period in the collective key decomposition in Section 4.2.2, and the results are therefore comparable.

Shared Period The first option is to decompose the columns such that they all share the same period, but each series has its scaling factors. To achieve this, we want to use a Kronecker decomposition where the scaling-matrix has a column for each series. The period-matrix has one single column that describes the shared period. Thus,

$$
D = \begin{bmatrix} s_{11}P & s_{12}P & \dots & s_{1n_s}P \\ s_{21}P & s_{22}P & \dots & s_{2n_s}P \\ \vdots & \vdots & \ddots & \vdots \\ s_{m_s1}P & s_{m_s2}P & \dots & s_{m_sn_s}P \end{bmatrix} = \begin{bmatrix} s_{11} & s_{12} & \dots & s_{1n_s} \\ s_{21} & s_{22} & \dots & s_{2n_s} \\ \vdots & \vdots & \ddots & \vdots \\ s_{m_s1} & s_{m_s2} & \dots & s_{m_sn_s} \end{bmatrix} \otimes \begin{bmatrix} p_{11} \\ p_{21} \\ \vdots \\ p_{m_p1} \end{bmatrix} = S \otimes P \ ,
$$

where the scaling-matrix $S \in \mathbb{R}^{m_s \times n_d}$ has one column for each series ($n_s = n_d$) and the period-matrix $P \in \mathbb{R}^{m_p \times 1}$ has a single column ($n_p = 1$).

Because the data we want to decompose together is the temperature data from different cities in Switzerland, it should be reasonable to assume that the temperature in all cities follows a similar pattern over a day. Therefore, we expect that sharing the period should enable us to compress the data better than if we decompose each series individually. The resulting RMSE vs. compression ratio curve is shown in Figure 4.10.

To be able to compare the collective and the individual compression and to make it comparable to our previous results, we calculated the reconstruction error only for the temperature data of Kloten ZH, while the other cities were only used for the decomposition. The compression ratio is calculated for the entire data set, including the data from the other cities.

As we can see, sharing the period does not seem to improve the compression, at least if we use the natural period of 1 day. This is because the period is very small compared to the scaling, which means that sharing the period does not significantly increase our compression, while it does increase the error. However, if we use 5 days, which is significantly longer, we can see that sharing the period does improve the compression. This suggests that sharing the period can improve the compression, but only if the period is large enough. Figure 4.11 shows the first two ranks of this decomposition.

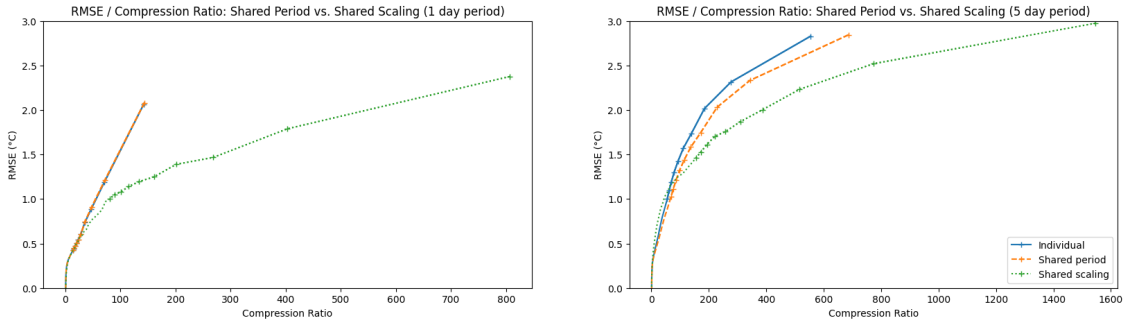


Figure 4.10: The approximation error for the temperature in Kloten ZH, when it is decomposed with the temperatures of other cities in Switzerland, sharing the period vs. sharing the scaling. The comparison is done for the natural period of 1 day in the left plot and 5 days in the right plot. The first ten ranks are marked with a plus sign.

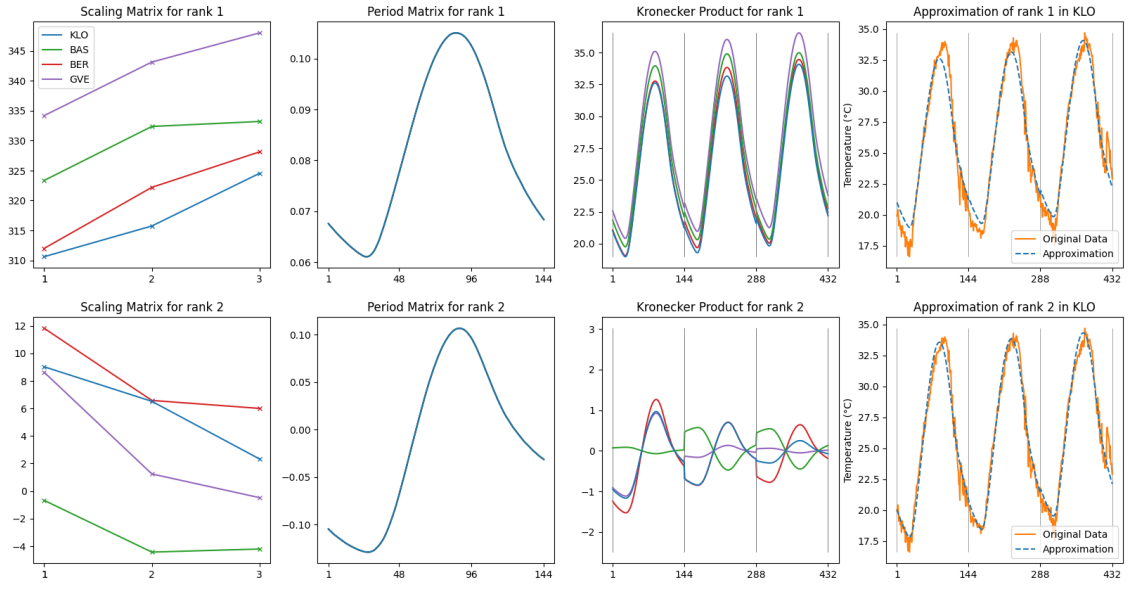


Figure 4.11: The first two ranks of the Kronecker decomposition with a shared period for four different locations in Switzerland. The locations are Kloten (KLO), Basel (BAS), Bern (BER) and Geneva (GVE).

Shared Scaling Sharing the scaling works similarly to sharing the period, but instead of having a single column in the period-matrix, we have a single column in the scaling-matrix. Thus,

$$D = \begin{bmatrix} s_{11}P \\ s_{21}P \\ \vdots \\ s_{m_s 1}P \end{bmatrix} = \begin{bmatrix} s_{11} \\ s_{21} \\ \vdots \\ s_{m_s 1} \end{bmatrix} \otimes \begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1n_p} \\ p_{21} & p_{22} & \cdots & p_{2n_p} \\ \vdots & \vdots & \ddots & \vdots \\ p_{m_p 1} & p_{m_p 2} & \cdots & p_{m_p n_p} \end{bmatrix} = S \otimes P \ ,$$

where the scaling-matrix $S \in \mathbb{R}^{m_s \times 1}$ has a single column ($n_s = 1$) and the period-matrix $P \in \mathbb{R}^{m_p \times n_d}$ has one column for each series ($n_p = n_d$).

In this case, we expect the different temperature curves to have their daily pattern, but that they behave similarly relative to each other. This could mean that if it is usually warmer in Geneva than in Zürich, this should be encoded in the different periods for the two cities, where we expect the average value of the period for Geneva to be larger than the average value of the period for Zürich. Note that they still share the same scaling, which means that we expect that if it is a warm day in Geneva, it is also a warm day in Zürich. The results for this decomposition are also shown in Figure 4.10.

Other than for the shared period, we can see that sharing the scaling does improve the compression significantly for the natural period of 1 day. This is because the scaling is much larger than the period, which means that sharing the scaling can significantly reduce the size of the compression. In Figure 4.12 we can see that we were right in our assumption that the periods are very similar, but some cities are usually warmer than others.

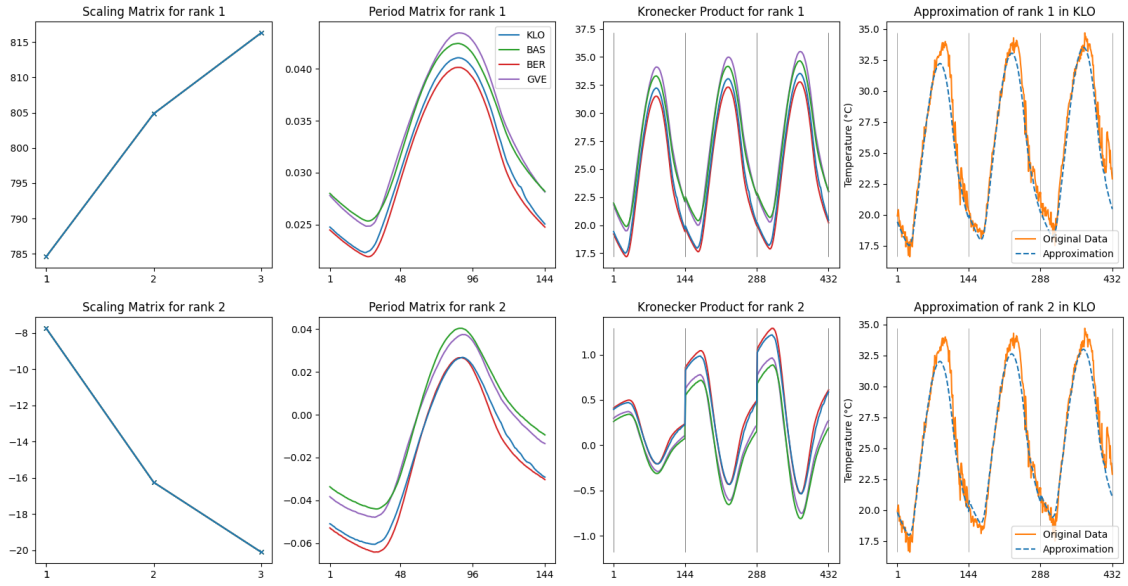


Figure 4.12: The first two ranks of the Kronecker decomposition with a shared scaling for four different locations in Switzerland. The locations are Kloten (KLO), Basel (BAS), Bern (BER) and Geneva (GVE).

Heterogeneous Data and Normalization

In this section, we want to decompose multiple time series with different value ranges. We will again look at meteorological data from MeteoSwiss but this time the location is fixed to Kloten

ZH and different measurements are taken at the same location. The measurements for Kloten ZH that we want to decompose together are shown in Table 4.1.

Measurement	**Unit**
Temperature 2m above ground	°C
Dew Point 2m above ground	°C
Gust Peak 10min max	m/s
Humidity 2m above ground	%
Precipitation	mm
Pressure at station level	hPa
Global Radiation	W/m^2
The temperature at 5cm above grass	°C
Wind Speed 10min mean	m/s

Table 4.1: The measurements that we want to decompose together. *Data Source: MeteoSwiss [Met]*

If we just decompose all of these measurements together, we will get a better approximation error for a given compression ratio for some of the measurements and a worse approximation error for others, compared to decomposing each measurement individually. The reason for this is that the measurements have very different value ranges. As described in Section 2.1, the Kronecker decomposition minimizes the Frobenius norm of the approximation error. If we have a column, that has very large absolute values compared to the other columns, the approximation error for this column will have a much larger impact on the Frobenius norm than the other measurements. This measurement will dominate the decomposition and will be approximated very well, while other measurements with comparably small values will be approximated worse.

To mitigate this problem, we need to normalize the data before decomposing it. The easiest way to normalize the data is to divide each measurement by its maximum absolute value, this way all measurements will have values in the range $[-1, 1]$. Of course, we can also use other normalization techniques, but we always have to be careful if we introduce a shift in the data, because as described in Section 4.4 this will change the Kronecker decomposition of each measurement. It is also much easier to recover the original data if we only use a scaling factor. To avoid confusion with the scaling factors in the scaling-matrix, we will call this scaling factor the normalization factor: c_{normal}. Taking the normalization factor for each measurement results in a row vector $\mathbf{c}_{normal} \in \mathbb{R}^{n_d}$. The length of this vector, n_d, is the number of measurements which is the number of columns in the data-matrix.

If we use a shared scaling for all measurements, each of the measurements will have a separate period-column in the period-matrix, which can directly be multiplied with the normalization factor after the decomposition. Thus,

$$D * \mathbf{c}_{normal} = \begin{bmatrix} d_{11}c_1, & d_{12}c_2, & \dots, & d_{1n_d}c_{n_d} \\ d_{21}c_1, & d_{22}c_2, & \dots, & d_{2n_d}c_{n_d} \\ \vdots & \vdots & \ddots & \vdots \\ d_{m_d1}c_1, & d_{m_d2}c_2, & \dots, & d_{m_dn_d}c_{n_d} \end{bmatrix} = \begin{bmatrix} s_{11}P * \mathbf{c}_{normal} \\ s_{21}P * \mathbf{c}_{normal} \\ \vdots \\ s_{m_s1}P * \mathbf{c}_{normal} \end{bmatrix} = S \otimes (P * \mathbf{c}_{normal})$$

where the $*$ operator is in this case the column-wise multiplication of a matrix M and the row vector $\mathbf{v}$. The same can be done for the scaling-matrix if we use a collective decomposition with a shared period.

If we use a normalization technique that involves shifting the data, for example, min-max-normalization or standardization, we cannot revert the normalization directly in the decomposed form. We will have to revert the shift while querying the data, which is the same problem that we have when we shift the data in Section 4.4. We will discuss how to do this in Section 5.3.6.

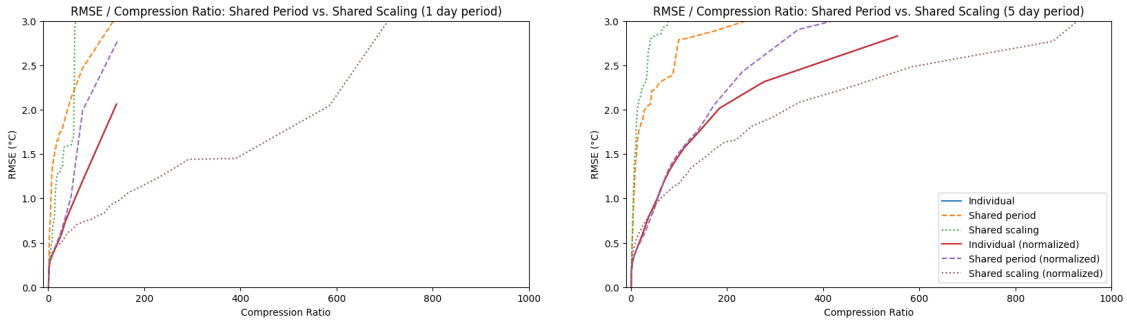


Figure 4.13: The collective decomposition of the measurements from Table 4.1. The measurements are normalized by dividing each measurement by its maximum absolute value. The error is calculated as the RMSE of the approximation of the temperature 2m above ground, to keep it comparable to the results in the previous sections. The plots for the individual decompositions are completely overlapping, because the Kronecker decomposition is scale invariant.

How well the collective decomposition works depends on how similar the patterns of the measurements are. In our case, we can see the results in Figure 4.13. The normalization has a significant effect on the approximation error. Only the normalized matrix with shared scaling has a better approximation error using the collective decomposition. The periods of the different measurements are too different to be approximated well by a shared period.

4.6 Further Experiments

In this section, we will show that the observations made in the previous sections generalize to other time series than the temperature in Kloten. For this purpose, we will use more measurements and locations from the MeteoSwiss dataset as well as a traffic dataset called UTD19 [UTD], which contains vehicle counts for many cities in the world.

We conclude that the observations made in the previous sections generalize to other time series. It is therefore possible to use the Kronecker decomposition to compress a wide range of time series but each time series must be analyzed individually because the optimal decomposition depends on the data.

In Section 4.6.1, we will look at the temperature in different cities in Switzerland. In Section 4.6.2, we will look at different measurements in Kloten. In Section 4.6.3, we will look at the traffic in Luzern.

4.6.1 MeteoSwiss: Temperatures in Switzerland

In the previous sections, we only looked at the approximation error for the temperature in Kloten. Figure 4.14, shows how the results generalize to other cities in Switzerland. The different temperature time series were normalized by dividing by the maximum temperature in the time series, as discussed in Section 4.5.2. We can see that the results for the other cities are similar to the results for Kloten.

In the first row, the period length is set to the natural period of 1 day. Sharing the period does not have a significant impact on the compression ratio but the periods are similar enough such that the error over the compression curve is almost identical to the individual decomposition. Only sharing of the scaling-column has a significant impact on the compression ratio and flattens the curve for high compression ratios. Compared to the averaging baseline, the compression is significantly better for all three approaches.

Using a period length of 5 days, the compression increase for the shared period compared to

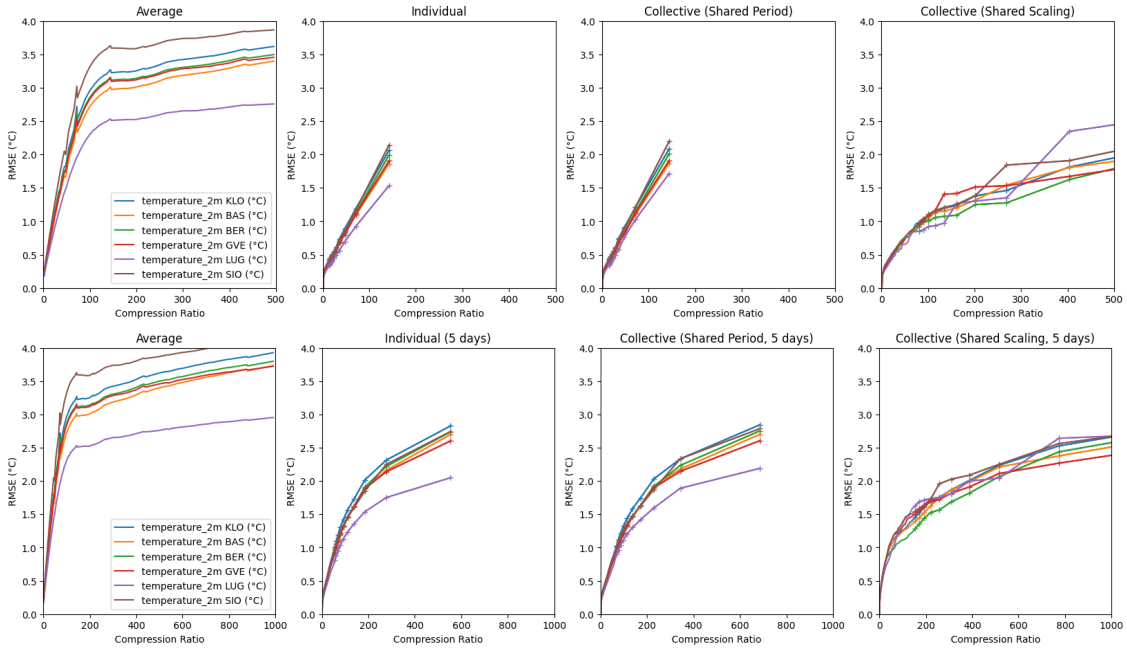


Figure 4.14: Plotted are the root mean square approximation errors (RMSEs) of the temperature in different cities in Switzerland. The top row is decomposed with a period length of 1 day and the bottom row with a period length of 5 days.

the individual decomposition becomes significant and therefore big enough to give an advantage for high compression ratios. This was expected from the results in Figure 4.10.

4.6.2 MeteoSwiss: Weather in Kloten

Figure 4.15, shows the approximation error for the different measurements in Kloten, as listed in Table 4.1. The different measurements were again normalized by dividing by the maximum value. The error is normalized as well to make it easier to compare the different measurements.

The results are again comparable to the result for the temperature in Kloten in Figure 4.13. Sharing the period is not a good idea in this case, because the periods are not similar enough and the decompression becomes generally worse and less consistent. Sharing the scaling-column seems to work especially well for the two temperature measurements, but not for the others, and should therefore be used with caution. It is more consistent than sharing the period and should be used especially if high compression ratios are desired.

4.6.3 UTD19: Traffic in Luzern

The UTD19 dataset contains traffic data for many cities in the world [UTD]. We chose the city of Luzern in Switzerland for our experiments, because there we have access to continuous data over one year for multiple locations in the city. The sensors in Luzern measure the number of cars passing by in a three-minute interval. This gives us a natural period of 480 three-minute intervals in one day because we expect the traffic to be periodic over a day. As for the MeteoSwiss dataset, the UTD19 dataset was prepared by interpolating any missing values using linear interpolation. Other than that, the data was not modified.

The first row in Figure 4.16, shows therefore the approximation error over the compression ratio for a period length of 1 day. The big difference to the MeteoSwiss dataset is that the natural period is larger than the square root of the number of datapoint, i.e. the period-columns are larger

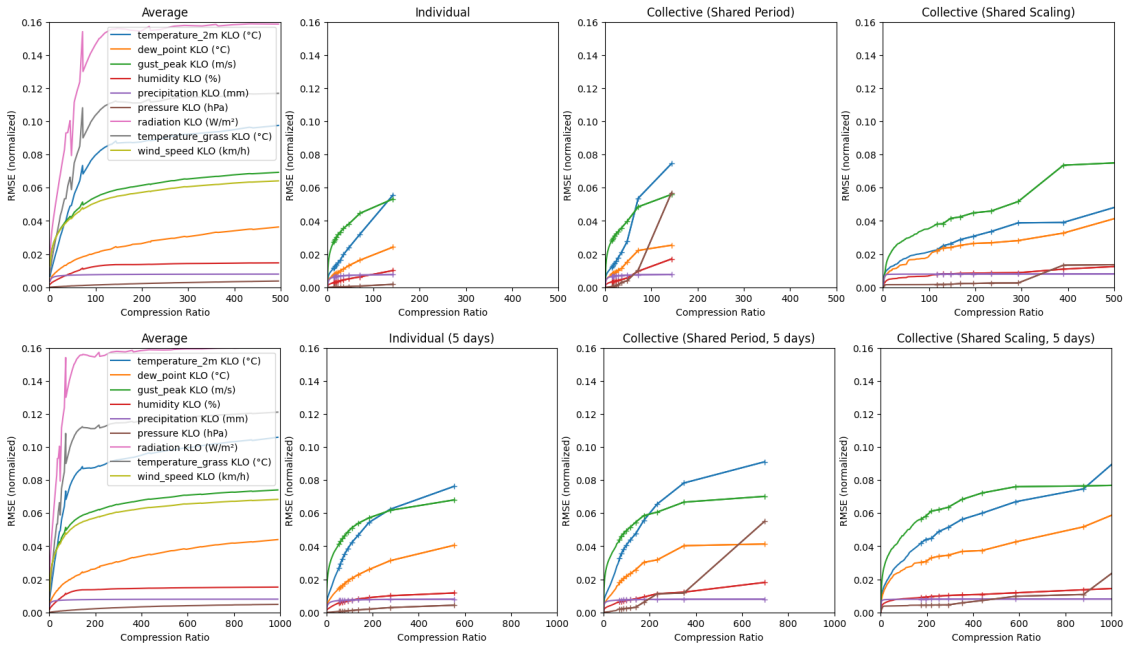


Figure 4.15: Plotted are the normalized root mean square approximation errors (RMSEs) of different measurements in Kloten. The top row is decomposed with a period length of 1 day and the bottom row with a period length of 5 days.

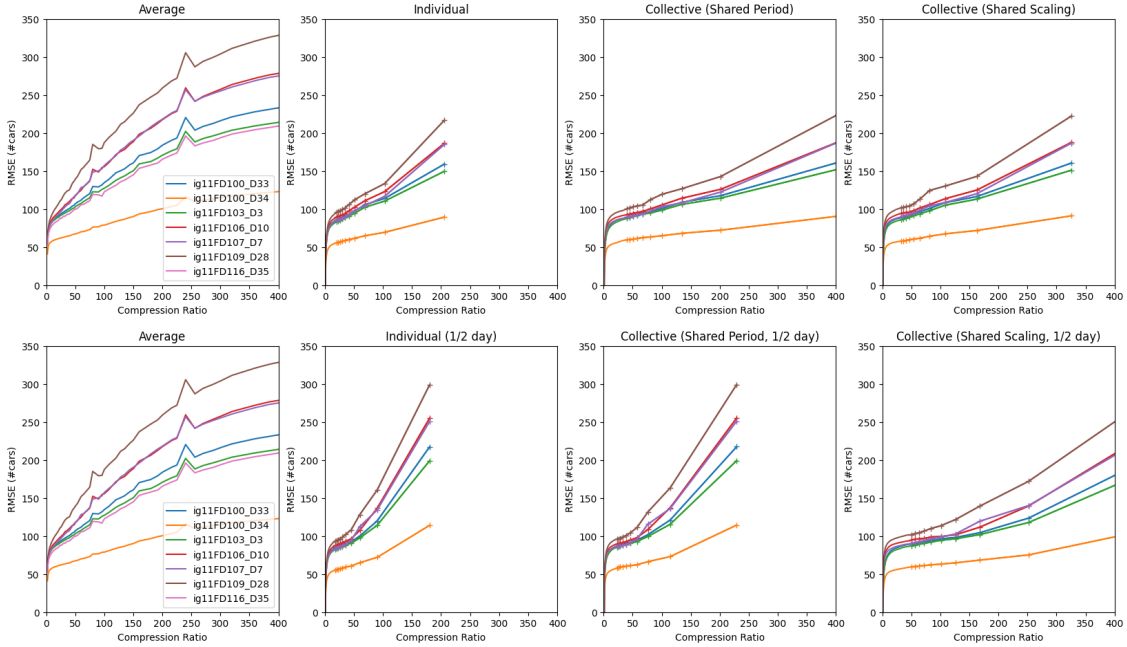


Figure 4.16: Plotted are the root mean square approximation errors (RMSEs) of the traffic in Luzern. The top row is decomposed with a period length of 1 day and the bottom row with a period length of 0.5 days. *Data Source: UTD19 [UTD]*

than the scaling-columns (480 vs. 365), which should generally be avoided. However, the periodic behavior of the traffic is strong enough such that the Kronecker decomposition is still significantly better than the averaging baseline. Both methods of collective decomposition bring an additional improvement over individual decomposition.

To get a smaller period, the second row in Figure 4.16 shows the approximation error for a period length of 0.5 days. The approximation becomes significantly worse because the period is no longer a multiple of the natural period, except for the decomposition with a shared scaling-column, where the advantage in compression ratio is large enough to compensate for the worse approximation.

Chapter 5

Query Processing in KroneDB

One of the main advantages of Kronecker decomposition is that we can compute a large number of relational queries directly on the decomposed data. Such queries may have operators such as selection, projection, join, and aggregates. In this chapter, we will discuss how different queries can be evaluated on the decomposed relations.

In Section 5.1 and Section 5.2, the selection and projection operators are discussed. In Section 5.3, we will look at the most common aggregates: sum, count, average, min, max and product. A special focus will be on the sum product, which is a generalization of the sum and is used in many data analysis and machine learning applications [Olt20]. In Section 5.4, it is shown how the KroneRelations fit into larger queries where they are joined with other relations. Finally, in Section 5.5, the efficiency of the different queries is tested on the MeteoSwiss dataset.

We conclude that the selection and projection operators can be run in one pass over the decomposed data with a performance decrease compared to the uncompressed data. For the aggregates, we show that it is not only possible but also more efficient to run them directly on the decomposed data. It is shown that the KroneRelations in a larger context can simply be substituted by the decomposed relations without any modifications to the query and other query optimizations can be applied as usual. The experiments confirm the theoretical results about the performance of the different queries.

5.1 Selection

In this section, we will discuss how selection queries can be executed directly on the decomposed data. For this purpose, we want to make a distinction between selection on the key columns, selection on the index column, and selection on the value columns. We will show how we can map a selection query on the original data to a selection query on the decomposed data.

For simplicity, we will only use a single key and a single value column in the SQL queries. The rank is assumed to be 1 and the rank column is omitted. The generalization to multiple key columns can be done by simply adding the additional key columns to the selection and join conditions. Generalizing to multiple value columns and ranks is also straightforward but requires a bit more notation which will be introduced in Section 5.3.1. We further assume that all key decompositions have the same period length m_p.

Selection on the key columns The selection of keys is the easiest case because we can simply select the rows that have the requested value as their key attribute. We generally assume that the key columns are categorical, therefore we only consider equality as a selection condition. A naive approach would be to join the decomposed relations and then select the rows with the requested key value on the join result. However, this is not the most efficient way, because we can select directly on the decomposed relations and avoid having keys in the join result which are not needed.

This simple optimization is usually done by the query optimizer but we will do it manually for the sake of clarity. Expressed in SQL, selecting all tuples with the key value "a", looks like this:

```
SELECT s.x, s.rid_s * {m_p} + p.rid_p AS rid, s.s * p.p AS d
FROM (
    SELECT *
    FROM scaling
    WHERE x = "a"
) AS s
JOIN (
    SELECT *
    FROM period
    WHERE x = "a"
) AS p
ON s.x = p.x;
```

where `{m_p}` is the period length. The `rid` attribute is calculated according to (4.1).

Selection on the index column The index column is not categorical, so we can use equality and inequality selection.

For inequality selection, it is not generally possible to optimize the naive approach of joining the decomposed relations and then selecting on the join result. Therefore, the SQL query to select all tuples with an `rid` smaller than 5 looks like this:

```
SELECT s.x, s.rid_s * {m_p} + p.rid_p AS rid, s.s * p.p AS d
FROM scaling s
JOIN period p
ON s.x = p.x
WHERE s.rid_s * {m_p} + p.rid_p < 5;
```

In the case of an equality selection, we can again optimize the query. By using the `rid` calculation form (4.1), we know that

$$rid_s = \left\lfloor \frac{rid - 1}{m_p} \right\rfloor + 1 \text{ and } rid_p = (rid - 1) \,\%\, m_p + 1,$$

where % is the modulo operator which returns the remainder of the division.

Assume that the period length $m_p = 2$ and we want to select the row with $rid = 5$. Then we know that $rid_s = 3$ and $rid_p = 1$ and we can write the query as

```
SELECT s.x, s.rid_s * {m_p} + p.rid_p AS rid, s.s * p.p AS d
FROM (
    SELECT *
    FROM scaling
    WHERE rid_s = 3
) AS s
JOIN (
    SELECT *
    FROM period
    WHERE rid_p = 1
) AS p
ON s.x = p.x;
```

This optimization cannot be done by the query optimizer because it requires knowledge about the Kronecker decomposition.

Selection on the value columns To select on the value columns, we have no choice but to join the decomposed relations and then select on the reconstructed value columns. An SQL query that selects all tuples with a d-value greater than 5 would be

```
SELECT s.x, s.rid_s * {m_p} + p.rid_p AS rid, s.s * p.p AS d
FROM scaling s
JOIN period p
ON s.x = p.x
WHERE s.s * p.p > 5;
```

5.2 Projection

In this section, we discuss how we can do a projection on the decomposed relations. This is a very simple operation, as we can simply project the columns of the decomposed relations that correspond to the columns in the data-relation that we want to project on. As we did in the previous section, we will show how to do a projection on key, index, and value columns separately. Also here, we will assume that the rank is 1 and the rank column is omitted.

Projection on the Key Columns Projecting on key columns is straightforward, as we can directly project on the key columns of the decomposed relations. Consider a KroneRelations with the key columns $x_1, \ldots, x_n$, where $n \geq 2$. Then, the SQL query to project on the key columns x_1 and x_2 is written as

```
SELECT s.x_1, s.x_2
FROM (
    SELECT x_1, x_2
    FROM scaling
) AS s
JOIN (
    SELECT x_1, x_2
    FROM period
) AS p
ON s.x_1 = p.x_1 AND s.x_2 = p.x_2;
```

If the goal is to get all unique combinations of the key values, then we can simply project on the key columns of the scaling- or period-relation, because both contain all unique combinations of the key values. Therefore,

```
SELECT DISTINCT x_1, x_2
FROM data;
```

is equivalent to

```
SELECT DISTINCT x_1, x_2
FROM scaling;
```

and

```
SELECT DISTINCT x_1, x_2
FROM period;
```

Projection on the Index Column Assume that we have a single key column x and a period length m_p. Projecting on the index column requires a join on the key columns to avoid a cross-product across different keys. Therefore is the only choice to join the decomposed relations on the key column and then project on the index column as follows:

```
SELECT s.rid_s * {m_p} + p.rid_p AS rid
FROM (
    SELECT x, rid_s
    FROM scaling
) AS s
JOIN (
    SELECT x, rid_p
    FROM period
) AS p
ON s.x = p.x;
```

Projection on the Value Columns Projecting on value columns is done analogously to the projection on the index column. Consider a KroneRelation with the value columns $d_1, \ldots, d_n$, which are decomposed into the scaling columns $s_1, \ldots, s_n$ and the period columns $p_1, \ldots, p_n$, using the individual column decomposition. Then, the SQL query to project on the value columns d_1 and d_2 can be written as

```
SELECT s_1 * p_1 AS d_1, s_2 * p_2 AS d_2
FROM (
    SELECT x, s_1, s_2
    FROM scaling
) AS s
JOIN (
    SELECT x, p_1, p_2
    FROM period
) AS p
ON s.x = p.x;
```

where `{s_i}` and `{p_j}` are the decomposed value columns that correspond to the data-column `d_f`.

5.3 Aggregates

In this section, we will look at the five most common aggregates: sum, count, average, min, and max. The focus will be on the sum aggregate over the product of multiplied columns, which is called sum product, in Section 5.3.5. In Section 5.3.7, we will look at the runtime complexity of the aggregates over the decomposed data. Finally, in Section 5.3.8, we will briefly discuss more complex aggregations, which are combinations of the discussed aggregates.

To keep track of the columns in the decomposed matrices, we introduce a special notation for the columns. The columns in S and P are indexed by the column index, i.e. $\mathbf{s}_1, \ldots, \mathbf{s}_{n_s}$ and $\mathbf{p}_1, \ldots, \mathbf{p}_{n_p}$. The columns in the data-matrix D are indexed by the column indices of the corresponding columns in the scaling and period matrices, i.e.

$$\mathbf{d}_{u,v} = \mathbf{s}_u \otimes \mathbf{p}_v, \text{ for all } u \in [n_s], v \in [n_p], \tag{5.1}$$

where n_s is the number of columns in the scaling-matrix and n_p is the number of columns in the period-matrix. This notation makes it easier to keep track of the columns in the decomposed matrices and provides a general mapping independent of the column decomposition strategy unlike the mappings in (4.2) and (4.3) which are specifically for the collective and individual column decomposition, respectively.

Thus, for the collective column decomposition, the columns are indexed as follows:

$$S \otimes P = \underbrace{\begin{bmatrix} s_{11} & s_{12} \\ s_{21} & s_{22} \\ s_{31} & s_{32} \end{bmatrix}}_{\mathbf{s}_1 \quad \mathbf{s}_2} \otimes \underbrace{\begin{bmatrix} p_{11} & p_{12} \\ p_{21} & p_{22} \end{bmatrix}}_{\mathbf{p}_1 \quad \mathbf{p}_2} = \underbrace{\left[\begin{array}{cc|cc} s_{11}p_{11} & s_{11}p_{12} & s_{12}p_{11} & s_{12}p_{12} \\ s_{11}p_{21} & s_{11}p_{22} & s_{12}p_{21} & s_{12}p_{22} \\ \hline s_{21}p_{11} & s_{21}p_{12} & s_{22}p_{11} & s_{22}p_{12} \\ s_{21}p_{21} & s_{21}p_{22} & s_{22}p_{21} & s_{22}p_{22} \\ \hline s_{31}p_{11} & s_{31}p_{12} & s_{32}p_{11} & s_{32}p_{12} \\ s_{31}p_{21} & s_{31}p_{22} & s_{32}p_{21} & s_{32}p_{22} \end{array}\right]}_{\mathbf{d}_{1,1} \quad \mathbf{d}_{1,2} \quad \mathbf{d}_{2,1} \quad \mathbf{d}_{2,2}} = D.$$

For the individual column decomposition, the columns are indexed accordingly as

$$S \circledast P = \underbrace{\begin{bmatrix} s_{11} & s_{12} \\ \vdots & \vdots \end{bmatrix}}_{\mathbf{s}_1 \quad \mathbf{s}_2} \circledast \underbrace{\begin{bmatrix} p_{11} & p_{12} \\ \vdots & \vdots \end{bmatrix}}_{\mathbf{p}_1 \quad \mathbf{p}_2} = \underbrace{\begin{bmatrix} s_{11}p_{11} & s_{12}p_{12} \\ \vdots & \vdots \end{bmatrix}}_{\mathbf{d}_{1,1} \quad \mathbf{d}_{2,2}} = D,$$

where $\circledast$ is the column-wise Kronecker product.

5.3.1 Sum

Consider the example above, summing up the first column, $\mathbf{d}_{1,1}$, of the data-matrix D gives us the following equation:

$$\sum_{d \in \mathbf{d}_{1,1}} d = s_{11}p_{11} + s_{11}p_{21} + s_{21}p_{11} + s_{21}p_{21} + s_{31}p_{11} + s_{31}p_{21},$$

where $\sum_{d \in \mathbf{d}_{1,1}} d$ is the sum of all elements in the column vector $\mathbf{d}_{1,1}$. We can rearrange the equation to get

$$\sum_{d \in \mathbf{d}_{1,1}} d = (s_{11} + s_{21} + s_{31})(p_{11} + p_{21}),$$

which is the same as the sum of the first column, $\mathbf{s}_1$ of the scaling matrix S, times the sum of the first column, $\mathbf{p}_1$ of the period-matrix P:

$$\sum_{d \in \mathbf{d}_{1,1}} d = \sum_{s \in \mathbf{s}_1} s \cdot \sum_{p \in \mathbf{p}_1} p.$$

In general, we can say that the sum over a column in D can be expressed as the product of the individual sums over the corresponding columns in S and P:

$$\sum_{d \in \mathbf{d}_{u,v}} d = \sum_{s \in \mathbf{s}_u} s \cdot \sum_{p \in \mathbf{p}_v} p. \tag{5.2}$$

This seems to be a very simple and efficient way to calculate the sum of a column in the original data-matrix, but we have not considered higher-rank decompositions yet. We define D^k as the rank-k approximation of the data-matrix D. Further are D^r, S^r, and P^r defined such that

$$D_k = \sum_{r=1}^{k} D^r = \sum_{r=1}^{k} S^r \otimes P^r$$

and

$$\mathbf{d}_{k|u,v} = \sum_{r=1}^{k} \mathbf{d}^r_{u,v} = \sum_{r=1}^{k} \mathbf{s}^r_u \otimes \mathbf{p}^r_v,$$

where $\mathbf{d}_{k|u,v}$ is the k-rank approximation of the column vector $\mathbf{d}_{u,v}$. To avoid too many subscripts, we will avoid using the subscript k for the k-rank approximation as it is usually clear from the context.

The calculation of the sum over $\mathbf{d}_{1,1}$ for rank k is therfore given by the equation

$$\sum_{d \in \mathbf{d}_{1,1}} d = (s_{11}^1 + s_{21}^1 + s_{31}^1)(p_{11}^1 + p_{21}^1) + \cdots + (s_{11}^k + s_{21}^k + s_{31}^k)(p_{11}^k + p_{21}^k),$$

which can be rewritten as

$$\sum_{d \in \mathbf{d}_{1,1}} d = \sum_{r=1}^{k} \left(\sum_{s \in \mathbf{s}_1^r} s \cdot \sum_{p \in \mathbf{p}_1^r} p \right).$$

In general, we can say that the sum over a column in the data-matrix can be expressed as the product of the individual sums over the corresponding columns in the scaling and period matrices, summed over all ranks:

$$\sum_{d \in \mathbf{d}_{u,v}} d = \sum_{r=1}^{k} \left(\sum_{s \in \mathbf{s}_u^r} s \cdot \sum_{p \in \mathbf{p}_v^r} p \right). \tag{5.3}$$

Summing over the column $d_{1,1}$ of the data-relation $\mathbf{D}$ for each key x can be expressed as the following SQL query:

```
SELECT x, SUM(d_11)
FROM data
GROUP BY x;
```

Using the result in (5.3) we can rewrite this query to aggregate over the decomposed relations $\mathbf{S}$ and $\mathbf{P}$ as follows:

```
SELECT x, SUM(s.sum_s * p.sum_p)
FROM (
    SELECT x, r, SUM(s_1) AS sum_s
    FROM scaling
    GROUP BY x, r
) AS s
JOIN (
    SELECT x, r, SUM(p_1) AS sum_p
    FROM period
    GROUP BY x, r
) AS p
ON s.x = p.x AND s.r = p.r
GROUP BY x;
```

where we sum over the columns s_1 and p_1 first and then take the product of the sums (`s.sum_s * p.sum_p`), before summing over all ranks to get the final sum for each key x.

5.3.2 Count

The count of a column in the original data-matrix is just the number of rows of the matrix. In our example, we have the data-matrix $D \in \mathbb{R}^{m_d \times n_d}$, where $m_d = 6$ and $n_d = 4$. The count over any column in the data-matrix is $m_d = 6$, which is equal to $m_s \cdot m_p = 3 \cdot 2 = 6$ by definition. This holds for every rank k. Thus, the count over any column in the data-matrix is rank-independent.

The SQL query for the count for every key in $\mathbf{D}$ is

```
SELECT x, COUNT(*)
FROM data
GROUP BY x;
```

To calculate the count over the decomposed relations **S** and **P**, we need to fix the rank because as mentioned above, the count is rank-independent. The SQL query for the count for every key over the decomposed relations **S** and **P** is therefore

```
SELECT x, count_s * count_p
FROM (
    SELECT x, COUNT(*) AS count_s
    FROM scaling
    WHERE r = 1
    GROUP BY x
) AS s
JOIN (
    SELECT x, COUNT(*) AS count_p
    FROM period
    WHERE r = 1
    GROUP BY x
) AS p
ON s.x = p.x;
```

where the rank is fixed to 1.

5.3.3 Average

The average is just the sum divided by the count. Therefore, the calculation of the average over any column vector $\mathbf{d}_{u,v}$ in the data-matrix D is given by

$$\frac{\sum_{d\in\mathbf{d}_{u,v}} d}{m_d} = \frac{\sum_{r=1}^{k}\left(\sum_{s\in\mathbf{s}_u^r} s \cdot \sum_{p\in\mathbf{p}_v^r} p\right)}{m_s \cdot m_p} = \sum_{r=1}^{k} \frac{\sum_{s\in\mathbf{s}_u^r} s}{m_s} \cdot \frac{\sum_{p\in\mathbf{p}_v^r} p}{m_p}. \tag{5.4}$$

This rewriting shows that the average of the data column $\mathbf{d}_{u,v}$ can be calculated by first calculating the average of $\mathbf{s}_u^r$ and $\mathbf{p}_v^r$ and then multiplying the two averages before summing over all ranks.

We can express the average over the data-relation **D** in SQL as

```
SELECT x, AVG(d_11)
FROM data
GROUP BY x;
```

This query can be rewritten to use the decomposition using the rewriting from (5.4) to get

```
SELECT x, SUM(avg_s * avg_p)
FROM (
    SELECT x, r, AVG(s_1) AS avg_s
    FROM scaling
    GROUP BY x, r
)
JOIN (
    SELECT x, r, AVG(p_1) AS avg_p
    FROM period
    GROUP BY x, r
)
ON s.x = p.x AND s.r = p.r
GROUP BY x;
```

5.3.4 Min, Max

Consider again our example from Section 5.3. The goal is to find the minimum of the first data column $\mathbf{d}_{1,1}$. Recall that the first column is the result of the following Kronecker product: $\mathbf{d}_{1,1} = \mathbf{s}_1 \otimes \mathbf{p}_1$. Thus, the minimum is calculated by

$$\min(\mathbf{d}_{1,1}) = min(s_{11}p_{11}, s_{11}p_{21}, s_{21}p_{11}, s_{21}p_{21}, s_{31}p_{11}, s_{31}p_{21}). \tag{5.5}$$

By definition of the Kronecker product, every element of $\mathbf{s}_1$ is multiplied with every element of $\mathbf{p}_1$ to form $\mathbf{d}_{1,1}$. In case there are no negative values in $\mathbf{s}_1$ or $\mathbf{p}_1$ the minimum of $\mathbf{d}_{1,1}$ is the minimum of $\mathbf{s}_1$ multiplied with the minimum of $\mathbf{p}_1$. Because there are negative values by construction of the Kronecker decomposition, this needs to be handled by including the maximum values of $\mathbf{s}_1$ and $\mathbf{p}_1$. Thus,

$$\min(\mathbf{d}_{1,1}) = \min(\min(\mathbf{s}_1) \cdot \min(\mathbf{p}_1), \min(\mathbf{s}_1) \cdot \max(\mathbf{p}_1), \max(\mathbf{s}_1) \cdot \min(\mathbf{p}_1), \max(\mathbf{s}_1) \cdot \max(\mathbf{p}_1)), \tag{5.6}$$

will capture the cases where $\mathbf{s}_1$ or $\mathbf{p}_1$ have negative values. The cases $\min(\mathbf{s}_1) \cdot \max(\mathbf{p}_1)$ and $\max(\mathbf{s}_1) \cdot \min(\mathbf{p}_1)$ cover that the smallest negative value multiplied by the largest positive value results in the smallest negative value overall and the last case, $\max(\mathbf{s}_1) \cdot \max(\mathbf{p}_1)$, covers the case where both $\mathbf{s}_1$ and $\mathbf{p}_1$ only have negative values. For the maximum, we can use the same reasoning to get

$$\max(\mathbf{d}_{1,1}) = \max(\min(\mathbf{s}_1) \cdot \min(\mathbf{p}_1), \min(\mathbf{s}_1) \cdot \max(\mathbf{p}_1), \max(\mathbf{s}_1) \cdot \min(\mathbf{p}_1), \max(\mathbf{s}_1) \cdot \max(\mathbf{p}_1)).$$

The minimum query for the data-relation $\mathbf{D}$ is given by

```
SELECT x, MIN(d_11)
FROM D
GROUP BY x;
```

For the query over the decomposition, we use the `LEAST` function which is implemented in most common database systems to get the following query corresponding to (5.6):

```
SELECT x, LEAST(
    min_s * min_p, min_s * max_p, max_s * min_p, max_s * max_p
)
FROM (
    SELECT x, MIN(s_1) AS min_s, MAX(s_1) AS max_s
    FROM S
    WHERE r = 1
    GROUP BY x
) JOIN (
    SELECT x, MIN(p_1) AS min_p, MAX(p_1) AS max_p
    FROM P
    WHERE r = 1
    GROUP BY x
);
```

The maximum query is analogous to the minimum query.

Rank-k The discussion in the previous paragraph only discussed the minimum and maximum of rank 1. Given rank $k = 2$, we can write the minimum as

$$\min(\mathbf{d}_{1,1}) = \min((s^1_{11}p^1_{11} + s^2_{11}p^2_{11}), (s^1_{11}p^1_{21} + s^2_{11}p^2_{21}), \ldots, (s^1_{31}p^1_{21} + s^2_{31}p^2_{21})).$$

To optimize this query, we need to consider that the second rank might change the position of the minimum value in $\mathbf{d}_{1,1}$. It is possible to use Fagin's algorithm or the threshold algorithm [Fag02]

to compute the extrema over multiple ranks without reconstruction of the data-matrix. Initial experiments showed that this approach seems to be slower than simply reconstructing the data-matrix and then computing the extrema over the reconstruction. We assume that the extrema of rank 1 are sufficiently good approximations for the extrema of rank k because rank 1 encodes the most important information of the data-matrix.

5.3.5 Sum Product

This section starts with a generalization of the sum and the dot product, the sum product, which is the elementwise multiplication of one or more vectors of equal length followed by a sum. This sum product is used to express in linear algebra what to do when summing over a product of columns in the data-matrix D.

Consider the data-matrix D as a set of column vectors $\mathbf{d}_1, \ldots, \mathbf{d}_{n_d} \in \mathbb{R}^{m_d}$. The goal is to take the sum product over a subset of the data-columns $D_l = \mathbf{d}_1, \ldots, \mathbf{d}_l \subseteq D$.

In SQL, this sum product is expressed as:

```
SELECT SUM(d_1 * ... * d_l)
FROM D;
```

where `d_1`, ..., `d_l` are the columns of D_l.

We use the following notation for the sum product in linear algebra:

$$\langle \mathbf{d}_1, \ldots, \mathbf{d}_l \rangle = \sum_{i=1}^{m_d} \mathbf{d}_1[i] \cdot \ldots \cdot \mathbf{d}_j[i],$$

where $\mathbf{d}_1, \ldots, \mathbf{d}_l$ are the column vectors in the sum product, m_d is the height of the data-matrix D and therefore the length of the column vectors, and $\mathbf{d}[i]$ is the i-th element of the vector $\mathbf{d}$. This notation is used to describe the sum product in the rest of this thesis.

An Example

Consider again the example introduced in Section 5.3. Calculating the sum product over the first two columns of the data-matrix D is done as follows:

$$\langle \mathbf{d}_{1,1}, \mathbf{d}_{1,2} \rangle = s_{11}p_{11} \cdot s_{11}p_{12} + s_{11}p_{21} \cdot s_{11}p_{22} + \cdots + s_{31}p_{11} \cdot s_{31}p_{12} + s_{31}p_{21} \cdot s_{31}p_{22}.$$

We can rearrange this equation to get

$$\langle \mathbf{d}_{1,1}, \mathbf{d}_{1,2} \rangle = (s_{11} \cdot s_{11} + s_{21} \cdot s_{21} + s_{31} \cdot s_{31})(p_{11} \cdot p_{12} + p_{21} \cdot p_{22}),$$

which is the product of the sum product over the scaling and period-matrices and can therefore be written as

$$\langle \mathbf{d}_{1,1}, \mathbf{d}_{1,2} \rangle = \langle \mathbf{s}_1, \mathbf{s}_2 \rangle \cdot \langle \mathbf{p}_1, \mathbf{p}_2 \rangle.$$

To get an example for higher ranks, we calculate the sum product over the first two columns of D for rank 2:

$$\langle \mathbf{d}_{1,1}, \mathbf{d}_{1,2} \rangle = (s^1_{11}p^1_{11} + s^2_{11}p^2_{11})(s^1_{11}p^1_{12} + s^2_{11}p^2_{12}) + \cdots + (s^1_{31}p^1_{21} + s^2_{31}p^2_{21})(s^1_{31}p^1_{22} + s^2_{31}p^2_{22}).$$

We can again rearrange this to get

$$\begin{aligned}
\langle \mathbf{d}_{1,1}, \mathbf{d}_{1,2} \rangle = {} & (s^1_{11}s^1_{11} + s^1_{21}s^1_{21} + s^1_{31}s^1_{31})(p^1_{11}p^1_{12} + p^1_{21}p^1_{22}) \\
& + (s^1_{11}s^2_{11} + s^1_{21}s^2_{21} + s^1_{31}s^2_{31})(p^1_{11}p^2_{12} + p^1_{21}p^2_{22}) \\
& + (s^2_{11}s^1_{11} + s^2_{21}s^1_{21} + s^2_{31}s^1_{31})(p^2_{11}p^1_{12} + p^2_{21}p^1_{22}) \\
& + (s^2_{11}s^2_{11} + s^2_{21}s^2_{21} + s^2_{31}s^2_{31})(p^2_{11}p^2_{12} + p^2_{21}p^2_{22}),
\end{aligned}$$

and use the sum product notation to write this as

$$\begin{aligned}\langle \mathbf{d}_{1,1}, \mathbf{d}_{1,2} \rangle = &\langle \mathbf{s}_1^1, \mathbf{s}_2^1 \rangle \cdot \langle \mathbf{p}_1^1, \mathbf{p}_2^1 \rangle \\ &+ \langle \mathbf{s}_1^1, \mathbf{s}_2^2 \rangle \cdot \langle \mathbf{p}_1^1, \mathbf{p}_2^2 \rangle \\ &+ \langle \mathbf{s}_1^2, \mathbf{s}_2^1 \rangle \cdot \langle \mathbf{p}_1^2, \mathbf{p}_2^1 \rangle \\ &+ \langle \mathbf{s}_1^2, \mathbf{s}_2^2 \rangle \cdot \langle \mathbf{p}_1^2, \mathbf{p}_2^2 \rangle.\end{aligned}$$

We will show how to calculate the sum product over the Kronecker decomposition for an arbitrary number of columns and ranks in the next section.

The Sum Product over the Kronecker decomposition

In this section, we will show how to calculate the sum product over two types of Kronecker decompositions. The first type is the individual column Decomposition, where all column vectors are decomposed individually. The second type is the collective column Decomposition, where the data-matrix D is decomposed as a whole.

Individual Column Decomposition We define the individual column Decomposition of rank 1 for a set of vectors $\{\mathbf{d}_1, \ldots, \mathbf{d}_l\} \in \mathbb{R}^{m_d}$ as two sets of vectors $\{\mathbf{s}_1, \ldots, \mathbf{s}_l\} \in \mathbb{R}^{m_s}, \{\mathbf{p}_1, \ldots, \mathbf{p}_l\} \in \mathbb{R}^{m_p}$ such that

$$\mathbf{d}_i = \mathbf{s}_i \otimes \mathbf{p}_i \text{ for all } i \in [l]. \tag{5.7}$$

By definition of the Kronecker Product, it holds that $m_d = m_s \cdot m_p$.
The sum product over the individual column Decomposition of rank 1 is given by the equation

$$\begin{aligned}\langle \mathbf{d}_1, \ldots, \mathbf{d}_l \rangle &= \sum_{i=1}^{m_d} \mathbf{d}_1[i] \cdot \ldots \cdot \mathbf{d}_l[i] && \text{definition of the sum product} \\ &= \sum_{j=1}^{m_s} \sum_{k=1}^{m_p} \mathbf{s}_1[j] \cdot \mathbf{p}_1[k] \cdot \ldots \cdot \mathbf{s}_l[j] \cdot \mathbf{p}_l[k] && \text{definition of the Kronecker Product} \\ &= \sum_{j=1}^{m_s} \mathbf{s}_1[j] \cdot \ldots \cdot \mathbf{s}_l[j] \cdot \sum_{k=1}^{m_p} \mathbf{p}_1[k] \cdot \ldots \cdot \mathbf{p}_l[k] && \text{distribution of the sum} \\ &= \langle \mathbf{s}_1, \ldots, \mathbf{s}_l \rangle \cdot \langle \mathbf{p}_1, \ldots, \mathbf{p}_l \rangle. && \text{definition of the sum product}\end{aligned} \tag{5.8}$$

We can extend this to the individual column Decomposition of rank k for a set of vectors $\{\mathbf{d}_1, \ldots, \mathbf{d}_l\} \in \mathbb{R}_d^m$. We define the individual column Decomposition of rank k as two sets of vectors $\{\mathbf{s}_1^1, \ldots, \mathbf{s}_l^k\} \in \mathbb{R}^{m_s}, \{\mathbf{p}_1^1, \ldots, \mathbf{p}_l^k\} \in \mathbb{R}^{m_p}$ such that

$$\mathbf{d}_i = \sum_{r=1}^{k} \mathbf{s}_i^r \otimes \mathbf{p}_i^r \text{ for all } i \in [l].$$

We further define $\mathbf{d}_i^r = \mathbf{s}_i^r \otimes \mathbf{p}_i^r$ for all $i \in [l], r \in [k]$, such that

$$\mathbf{d}_i = \sum_{r=1}^{k} \mathbf{d}_i^r \text{ for all } i \in [l]. \tag{5.9}$$

The sum product over the individual column Decomposition of rank k is therefore given by

the equation

$$
\begin{aligned}
\langle \mathbf{d}_1, \dots, \mathbf{d}_l \rangle &= \sum_{i=1}^{m_d} \mathbf{d}_1[i] \cdot \ldots \cdot \mathbf{d}_l[i] && \text{definition of the sum product} \\
&= \sum_{i=1}^{m_d} \left(\sum_{r=1}^{k} \mathbf{d}_1^r[i] \cdot \ldots \cdot \sum_{r=1}^{k} \mathbf{d}_l^r[i] \right) && \text{definition in (5.9)} \quad (5.10) \\
&= \sum_{i=1}^{m_d} \sum_{(r_1,\dots,r_l) \in [k]^l} \mathbf{d}_1^{r_1}[i] \cdot \ldots \cdot \mathbf{d}_l^{r_l}[i] && \text{distribution of the sum} \\
&= \sum_{(r_1,\dots,r_l) \in [k]^l} \left(\sum_{i=1}^{m_d} \mathbf{d}_1^{r_1}[i] \cdot \ldots \cdot \mathbf{d}_l^{r_l}[i] \right) && \text{distribution of the sum} \\
&= \sum_{(r_1,\dots,r_l) \in [k]^l} \langle \mathbf{d}_1^{r_1}, \dots, \mathbf{d}_l^{r_l} \rangle && \text{definition of the sum product} \\
&= \sum_{(r_1,\dots,r_l) \in [k]^l} \langle \mathbf{s}_1^{r_1}, \dots, \mathbf{s}_l^{r_l} \rangle \cdot \langle \mathbf{p}_1^{r_1}, \dots, \mathbf{p}_l^{r_l} \rangle. && \text{from rank 1 (5.8)} \quad (5.11)
\end{aligned}
$$

This means that the sum product over the columns in D can be calculated by first calculating the sum product over the corresponding columns in S and P and multiplying the results, as we have seen for rank 1. However, this is done for every possible combination of ranks $r_1, \dots, r_l$ and then summed up.

Collective column Decomposition In the previous paragraph, we showed how to calculate the sum product over the individual column Decomposition. In this paragraph, we will show how the sum product over the collective column decomposition is calculated in the same way.

The collective column Decomposition of rank 1 for a matrix $D \in \mathbb{R}^{m_d \times n_d}$ is defined as two matrices $S \in \mathbb{R}^{m_s \times n_s}, P \in \mathbb{R}^{m_s \times n_s}$ such that

$$D = S \otimes P.$$

By definition of the Kronecker Product, it holds that $m_d = m_s \cdot m_p$ and $n_d = n_s \cdot n_p$. Using the column notation introduced in Section 5.3, we can write the collective column Decomposition as

$$\mathbf{d}_{u_i,v_i} = \mathbf{s}_{u_i} \otimes \mathbf{p}_{v_i} \text{ for all } i \in [l].$$

This fits (5.7) and the same computations can be applied to the collective column decomposition to obtain

$$\langle \mathbf{d}_{u_1,v_1}, \dots, \mathbf{d}_{u_l,v_l} \rangle = \langle \mathbf{s}_{u_1}, \dots, \mathbf{s}_{u_l} \rangle \cdot \langle \mathbf{p}_{v_1}, \dots, \mathbf{p}_{v_l} \rangle.$$

Extending this to the collective column Decomposition of rank k for a matrix $D \in \mathbb{R}^{m_d \times n_d}$, we define the collective column Decomposition of rank k as two sets of matrices $\{S^1, \dots, S^k\} \in \mathbb{R}^{m_s \times n_s}, \{P^1, \dots, P^k\} \in \mathbb{R}^{m_p \times n_p}$ such that

$$D_k = \sum_{r=1}^{k} S^r \otimes P^r.$$

Again reusing the computation from the individual column decomposition results in

$$\langle \mathbf{d}_{u_1,v_1}, \dots, \mathbf{d}_{u_l,v_l} \rangle = \sum_{(r_1,\dots,r_l) \in [k]^l} \langle \mathbf{s}_{u_1}^{r_1}, \dots, \mathbf{s}_{u_l}^{r_l} \rangle \cdot \langle \mathbf{p}_{v_1}^{r_1}, \dots, \mathbf{p}_{v_l}^{r_l} \rangle. \quad (5.12)$$

The corresponding SQL query for the sum product over the Kronecker decomposition can be generated using Algorithm 1. Figure 11.1 in the appendix shows what such a query looks like for a sum product over two columns of a rank-2 Kronecker decomposition.

Algorithm 1 SQL query for the sum product over the Kronecker decomposition

Require: Key column names $\mathbf{x}$, sum product column names $\mathbf{s}$ and $\mathbf{p}$, decomposed relation names $\mathbf{S}$ and $\mathbf{P}$, rank k

for $(i, (r_1, \ldots, r_l)) \in (k^l, [k]^l)$ **do**

Self join $\mathbf{S}$ for each column in $\mathbf{s}_l$ and rank r in $(r_1, \ldots, r_l)$ as $\mathbf{S}_i$ and select the sum product over the value columns $\mathbf{s}^{(r_1,\ldots,r_l)}$ grouped by $\mathbf{x}$ as s'_i

Self join $\mathbf{P}$ for each column in $\mathbf{p}_l$ and rank r in $(r_1, \ldots, r_l)$ as $\mathbf{P}_i$ and select the sum product over the value columns $\mathbf{p}^{(r_1,\ldots,r_l)}$ grouped by $\mathbf{x}$ as p'_i

Join $\mathbf{S}_i$ and $\mathbf{P}_i$ on $\mathbf{x}$ as $\mathbf{D}_i$ and select the product $s'_i * p'_i$ as d_i

end for

Join all $\mathbf{D}_i$ on $\mathbf{x}$ and select the sum over all d_i as d

5.3.6 Recovering Shift in Aggregates

In the Sections 4.5.2 and 4.4 it was discussed that shifting the original data-matrix D by a constant value c can improve the approximation quality of the decomposition. In this section, we will discuss how to recover such a shift when aggregating over the decomposed data.

For the average, minimum, and maximum, recovering the shift is trivial. We can simply subtract the shift from the result of the aggregation, because

$$avg(\mathbf{d}_1 + c) = avg(\mathbf{d}_1) + c.$$

The same holds for the minimum and maximum.

In the case of a sum aggregation, the situation is slightly more complicated, because

$$\sum_{d \in (\mathbf{d}_1 + c)} d = \sum_{d \in \mathbf{d}_1} (d + c) = \left(\sum_{d \in \mathbf{d}_1} d \right) + m_d \cdot c.$$

This means that we need to subtract the shift multiplied by the number of elements in the column from the result of the sum.

For the sum product, the situation is even more complicated. We start by looking at the sum product of the columns $\mathbf{d}_{1,1}$ and $\mathbf{d}_{2,2}$. The sum product of these two columns is given by

$$\begin{aligned}
\langle \mathbf{d}_1 + c, \mathbf{d}_2 + c \rangle &= \sum_{i=1}^{m_d} (d_1[i] + c) \cdot (d_2[i] + c) \\
&= \sum_{i=1}^{m_d} d_1[i] \cdot d_2[i] + c \cdot \sum_{i=1}^{m_d} d_1[i] + c \cdot \sum_{i=1}^{m_d} d_2[i] + m_d \cdot c^2 \\
&= \langle \mathbf{d}_1, \mathbf{d}_2 \rangle + c \cdot \langle \mathbf{d}_1 \rangle + c \cdot \langle \mathbf{d}_2 \rangle + c^2 \cdot m_d.
\end{aligned}$$

For more than two columns we get

$$\begin{aligned}
&\langle \mathbf{d}_1 + c, \ldots, \mathbf{d}_l + c \rangle = \\
&\langle \mathbf{d}_1, \ldots, \mathbf{d}_l \rangle \\
&+ c \cdot \langle \mathbf{d}_1, \ldots, \mathbf{d}_{l-1} \rangle + c \cdot \langle \mathbf{d}_1, \ldots, \mathbf{d}_{l-2}, \mathbf{d}_l \rangle + \cdots + c \cdot \langle \mathbf{d}_2, \ldots, \mathbf{d}_l \rangle \\
&+ c^2 \cdot \langle \mathbf{d}_1, \ldots, \mathbf{d}_{l-2} \rangle + c^2 \cdot \langle \mathbf{d}_1, \ldots, \mathbf{d}_{l-3}, \mathbf{d}_{l-1} \rangle + \cdots + c^2 \cdot \langle \mathbf{d}_3, \ldots, \mathbf{d}_l \rangle \\
&\vdots \\
&+ c^{l-1} \cdot \langle \mathbf{d}_1 \rangle + c^{l-1} \cdot \langle \mathbf{d}_2 \rangle + \cdots + c^{l-1} \cdot \langle \mathbf{d}_{l-1} \rangle \\
&+ c^l \cdot m_d,
\end{aligned}$$

where l is the number of columns. This suggests that it should probably be avoided to shift the data-matrix D when using the sum product as an aggregate, especially when the number of columns is large.

5.3.7 Runtime Complexity Analysis

In this section, we compare the theoretical runtimes of calculating the different aggregates over the data-matrix $D \in \mathbb{R}^{m_d \times n_d}$ and the decomposed matrices $S \in \mathbb{R}^{m_s \times n_s}$ and $P \in \mathbb{R}^{m_p \times n_p}$.

Sum To calculate the sum over any column in D, we need m_d additions, which gives us a runtime complexity of $O(m_d)$. For the decomposed matrices, we need to calculate the sum over each column in S and P separately and then multiply the results. This leaves us with $m_s + m_p$ additions and one multiplication. This is done for each rank, which gives us a runtime complexity of $O(k(m_s + m_p))$, where k is the number of ranks.

Count The count over D is just the number of rows in D, which is m_d. The runtime complexity is therefore $O(1)$. For the decomposed matrices, the count is $m_s \cdot m_p$, which also gives us a runtime complexity of $O(1)$.

For relational databases, where the count is calculated by scanning the table, the runtime complexity is $O(m_d)$ for the count over the data-relation $\mathbf{D}$ and $O(m_s + m_p)$ over the decomposed relations $\mathbf{S}$ and $\mathbf{P}$.

Average For the average, which is just the sum divided by the count, we again get $O(m_d)$ and $O(k(m_s + m_p))$ for the average over the data-matrix D and the decomposed matrices S and P, respectively.

MinMax Since we only consider min and max for rank 1, we can calculate the min and max over the original data-matrix in $O(m_d)$ time and over the decomposed matrices in $O(m_s + m_p)$ time.

Sum Product For the sum product over D, we need $(l-1) \cdot m_d$ multiplications, where l is the number of columns in the sum product, for the element-wise multiplication of the columns and then $(m_d - 1)$ additions, for the sum of the resulting vector. This results in a runtime complexity of $O(l \cdot m_d)$.

For the decomposed matrices, we need to calculate the sum product for each combination in $[k]^l$ separately and then sum up the results. We need $(l-1) \cdot m_s$ multiplications and $m_s - 1$ additions for the sum product over S and $(l-1) \cdot m_p$ multiplications and $m_p - 1$ additions for the sum product over P. Overall, this gives us a runtime complexity of $O(k^l \cdot l \cdot (m_s + m_p))$.

The runtime of the sum product over the decomposed matrices is polynomial in the rank k, with a degree of l. Thus, if we aim to use the sum product efficiently, especially with a high number of columns, we should be cautious about selecting high ranks. As discussed in Section 4.3, if we want a specific accuracy or compression ratio for our approximation, we can choose if we want to reach it with a more balanced decomposition and a higher rank or with a less balanced decomposition and a lower rank. A decomposition is considered balanced if the heights of the scaling and period-matrices are similar. If we know, that we want to use the sum product, we should choose the less balanced decomposition with the lower rank to get a better runtime for the same accuracy or compression ratio.

5.3.8 Further Aggregates

Many more aggregations are just combinations of the ones we have already discussed. A few of these can be found in Table 5.1.

5.4 Joins

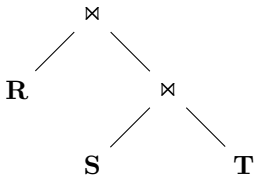
This section discusses the KroneRelations in the context of joins with other relations. Recall that the KroneRelation can only be joined on the key columns, which allows for a very simple plug-in

Aggregate	Formula
Covariance($\mathbf{d}_{1,1}, \mathbf{d}_{2,1}$)	$= \frac{<\mathbf{d}_{1,1},\mathbf{d}_{2,1}>}{m_d} - \frac{<\mathbf{d}_{1,1}>}{m_d} \cdot \frac{<\mathbf{d}_{2,1}>}{m_d}$
Variance($\mathbf{d}_{1,1}$)	$= \frac{<\mathbf{d}_{1,1},\mathbf{d}_{1,1}>}{m_d} - \frac{<\mathbf{d}_{1,1}>}{m_d} \cdot \frac{<\mathbf{d}_{1,1}>}{m_d}$
StandardDeviation($\mathbf{d}_{1,1}$)	$= \sqrt{\text{Variance}(\mathbf{d}_{1,1})}$
Correlation($\mathbf{d}_{1,1}, \mathbf{d}_{2,1}$)	$= \frac{\text{Covariance}(\mathbf{d}_{1,1},\mathbf{d}_{2,1})}{\text{StandardDeviation}(\mathbf{d}_{1,1}) \cdot \text{StandardDeviation}(\mathbf{d}_{2,1})}$
RootMeanSquare($d_{1,1}$)	$= \sqrt{\frac{<\mathbf{d}_{1,1},\mathbf{d}_{1,1}>}{m_d}}$
Skewness($d_{1,1}$)	$= \frac{\frac{<\mathbf{d}_{1,1},\mathbf{d}_{1,1},\mathbf{d}_{1,1}>}{m_d} - 3 \cdot \frac{<\mathbf{d}_{1,1},\mathbf{d}_{1,1}>}{m_d} \cdot \frac{<\mathbf{d}_{1,1}>}{m_d} - \frac{<\mathbf{d}_{1,1},\mathbf{d}_{1,1},\mathbf{d}_{1,1}>}{m_d^3}}{\text{StandardDeviation}(\mathbf{d}_{1,1})^3}$

Table 5.1: Further aggregates

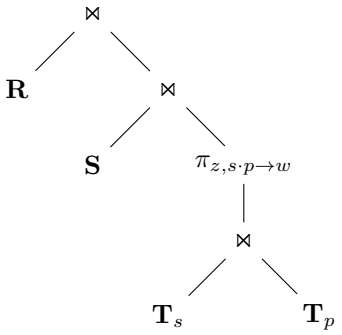
replacement of the KroneRelation with the decomposed relations in the context of joins.

Consider a join query of the form $\mathbf{R}(x, y) \bowtie \mathbf{S}(y, z) \bowtie \mathbf{T}(z, w)$ with the corresponding join tree and SQL query



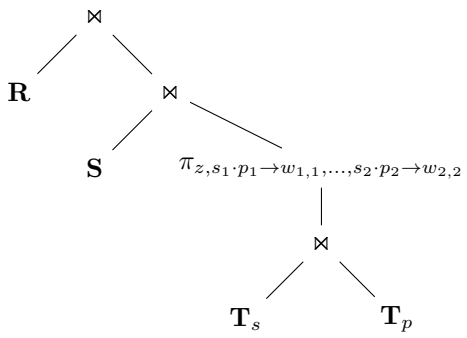

```
SELECT *
FROM R JOIN S ON R.y = S.y
    JOIN T ON S.z = T.z;
```

Further consider that $\mathbf{T}(z, w)$ is a KroneRelation with a single key column z and a single value column w. The index column is omitted to make the notation more readable. The decomposition of $\mathbf{T}$ is given by $\pi_{z,s \cdot p \to w}(\mathbf{T}_s(z, s) \bowtie \mathbf{T}_p(z, p))$. This definition can simply be plugged into the join query to replace $\mathbf{T}$ with the decomposed relations:




```
SELECT *
FROM R JOIN S ON R.y = S.y
    JOIN (
        SELECT z, s * p AS w
        FROM T_s JOIN T_p
        ON T_s.z = T_p.z
    ) AS T ON S.z = T.z;
```

To make it more interesting, consider the query $\mathbf{R}(x, y) \bowtie \mathbf{S}(y, z) \bowtie \mathbf{T}(z, w_{1,1}, w_{1,2}, w_{2,1}, w_{2,2})$, where $\mathbf{T}$ is a KroneRelation with four data-columns $w_{1,1}$, $w_{1,2}$, $w_{2,1}$, $w_{2,2}$. The decomposition of $\mathbf{T}$ is given by $\pi_{z,s_1 \cdot p_1 \to w_{1,1}, s_1 \cdot p_2 \to w_{1,2}, s_2 \cdot p_1 \to w_{2,1}, s_2 \cdot p_2 \to w_{2,2}}(\mathbf{T}_s(z, s_1, s_2) \bowtie \mathbf{T}_p(z, p_1, p_2))$:

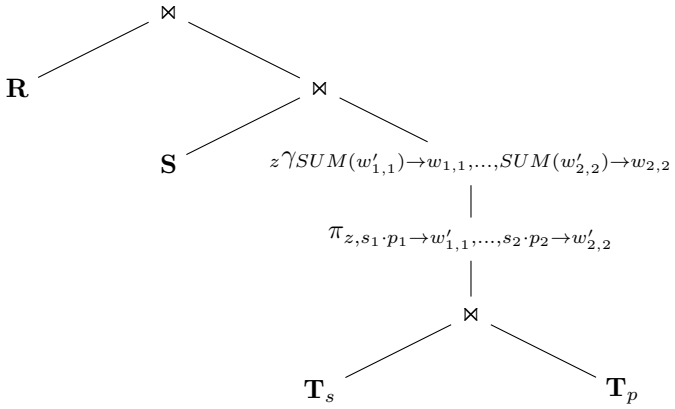



```
SELECT *
FROM R JOIN S ON R.y = S.y
    JOIN (
        SELECT z,
            s_1 * p_1 AS w_11,
            s_1 * p_2 AS w_12,
            s_2 * p_1 AS w_21,
            s_2 * p_2 AS w_22
        FROM T_s JOIN T_p
        ON T_s.z = T_p.z
    ) AS T ON S.z = T.z;
```

Finally, we need to introduce the rank k of the Kronecker decomposition. This means that the decomposed relations get an additional rank column r and the decomposition of $\mathbf{T}$ is given by

$$z\gamma_{SUM(w'_{1,1}) \to w_{1,1},\ldots,SUM(w'_{2,2}) \to w_{2,2}}(\pi_{z,s_1 \cdot p_1 \to w'_{1,1},\ldots,s_2 \cdot p_2 \to w'_{2,2}}(\mathbf{T}_s(z,r,s_1,s_2) \bowtie \mathbf{T}_p(z,r,p_1,p_2))).$$

This results in the new join tree and SQL query

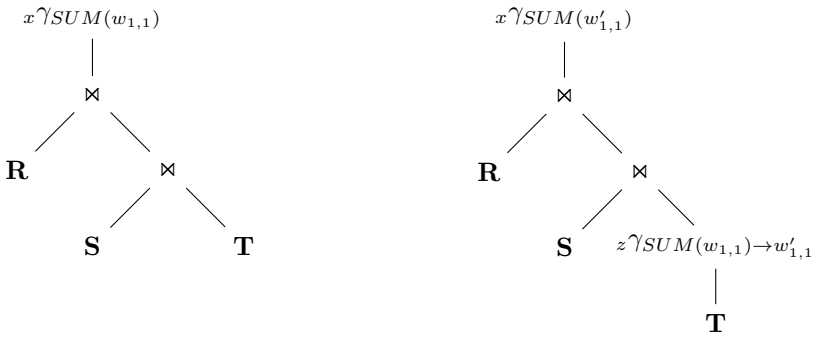



```
SELECT *
FROM R JOIN S ON R.y = S.y
    JOIN (
        SELECT z, SUM(s_1 * p_1) AS w_11,
            SUM(s_1 * p_2) AS w_12,
            SUM(s_2 * p_1) AS w_21,
            SUM(s_2 * p_2) AS w_22
        FROM T_s JOIN T_p
        ON T_s.z = T_p.z AND T_s.r = T_p.r
        GROUP BY z
    ) AS T ON S.z = T.z;
```

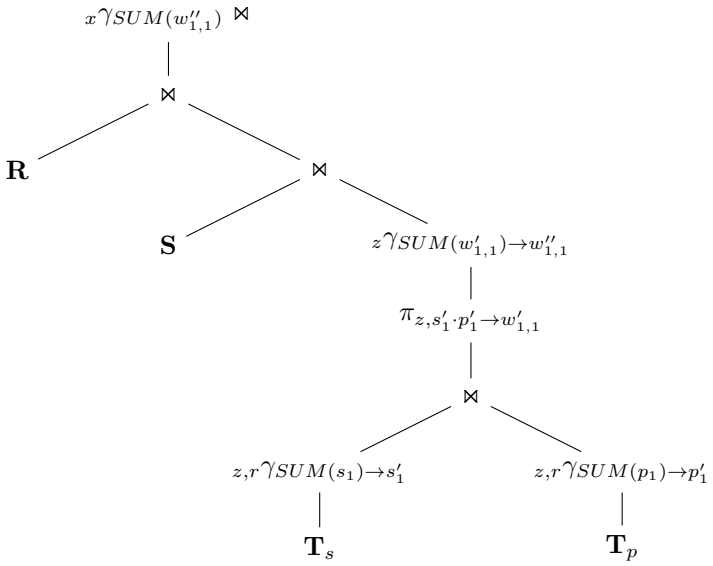
Aggregating over joins

For the join above, we can approximate the original data using our Kronecker decomposition. But we sacrifice execution time compared to running the join query on the original relation $\mathbf{T}$. Where we can gain execution time is when we aggregate over the decomposition.

Consider the query $({}_x\gamma_{SUM(w_{1,1})}\mathbf{R}(x,y) \bowtie \mathbf{S}(y,z) \bowtie \mathbf{T}(z,w))$ with the corresponding join tree. We can push the aggregation down the join tree, which is what a query optimizer would do:



Using the new query $({}_{x}\gamma_{SUM(w'_{1,1})}\mathbf{R}(x,y) \bowtie \mathbf{S}(y,z) \bowtie_{z} {}_{\gamma_{SUM(w_{1,1})\to w'_{1,1}}}\mathbf{T}(z,w))$, allows for a simple sum over a value column in the KroneRelation $\mathbf{T}$. We already know how to do this directly on the decomposed relations, giving us the new join tree and query:




```
SELECT x, SUM(w_11)
FROM R JOIN S ON R.y = S.y
    JOIN (
        SELECT z, SUM(s_1_sum * p_1_sum) AS w_11
        FROM (
            SELECT z, r, SUM(s_1) AS s_1_sum
            FROM T_s
            GROUP BY z, r
        ) AS Ts JOIN (
            SELECT z, r, SUM(p_1) AS p_1_sum
            FROM T_p
            GROUP BY z, r
        ) AS Tp
        ON Ts.z = Tp.z AND Ts.r = Tp.r
        GROUP BY z
    ) AS T ON S.z = T.z
GROUP BY x;
```

5.5 Experiments

In this section, we evaluate the performance of KroneDB. We compare the runtimes of KroneDuck over the decomposed relations with the runtime of DuckDB over the original relation. Recall that KroneDuck only creates the SQL queries and the decomposed relations. The actual execution of the queries is done by DuckDB in both cases. Note that the implementation of KroneDuck does not use a rank column in the decomposed relations but instead creates separate columns for each rank. Furthermore the performance of KronePy over the decomposed data compared to the performance of an equivalent Python implementation over the original data. Both Python implementations use NumPy arrays to store and process the data-, scaling- and period-matrices and take advantage of the tensor operations provided by NumPy.

System Setup The experiments were run on a Debian GNU/Linux 10 (buster) system with an Intel Xeon Silver 4241 CPU with 24 cores and 48 threads. The system has 200 GB of RAM. For DuckDB, the pip package `duckdb` version 0.9.0 was used. DuckDB was configured to use all 48 threads by executing `PRAGMA threads=48;` before each experiment. The runtime of DuckDB was measured using the Python APIs `explain('analyze')` method. For Python, version 3.9.16 was used with the NumPy pip package `numpy` version 1.26.0. NumPy was configured to use Intel Math Kernel Library (MKL) version 2023.1.0.

5.5.1 Selection

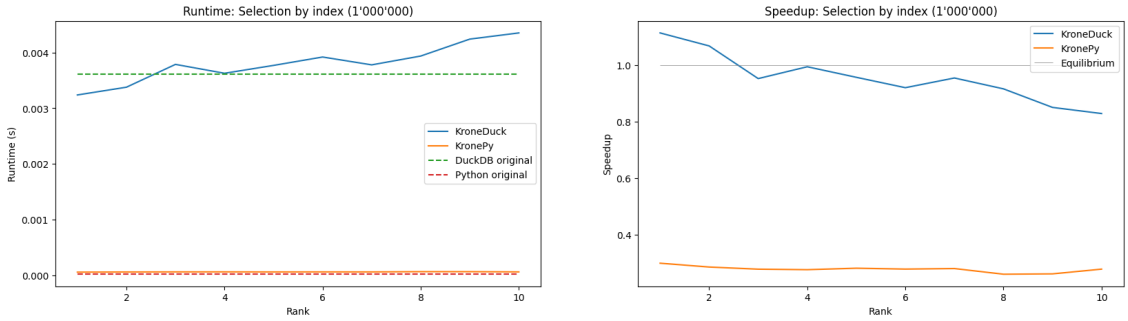


Figure 5.1: Runtime and speedup of a selection query on the index column. The selected index is $1'000'000$. The speedup of KronePy is compared to the baseline Python implementation and the speedup of KroneDuck is compared to DuckDB.

The dataset used for the selection queries is the temperatures for all six cities stacked on top of each other, with the station names (`KLO`, `BAS`, `BER`, `GVE`, `LUG`, `SIO`) as the keys. The period for the Kronecker decomposition is one day $m_p = 144$. The expectation is generally that the baseline NumPy and DuckDB implementations perform better than KronePy and KroneDuck because they do not need to reconstruct the original data from the decomposed relations. The expectations are met in both, selection by key and selection by value, as can be seen in Figures 11.2 and 11.3 in the appendix. Figure 5.1 shows the runtime and speedup of a selection query on the index column. In this case, KroneDuck outperforms DuckDB for the rank 1 and 2 decompositions. The reason is most likely that it is much faster to select a single row from the much smaller decomposed relations than to select a single row from the large original relation. Recall that in the KroneDuck implementation, each rank is stored in additional columns and not in additional rows. Thus, the decomposed relations have the same number of tuples independent of the rank. For less selective queries, the advantage of KroneDuck over DuckDB is mitigated because the overhead of reconstructing the original data becomes more significant. For the same reason is KroneDuck slower than DuckDB for higher ranks.

5.5.2 Projection

The same dataset as for the selection queries is used for the projection queries. The expectations for the projection queries are the same as for the selection queries. And there are no surprises in this case. The results can be seen in Figures 11.4, 11.5 and 11.6 in the appendix, but are not discussed in detail here. In summary, KronePy and KroneDuck are slower than the baseline Python and DuckDB implementations and they are worse for higher ranks if the projection is onto the value column because the original data needs to be reconstructed.

5.5.3 Sum

The same dataset as for the selection queries is used for the sum query.

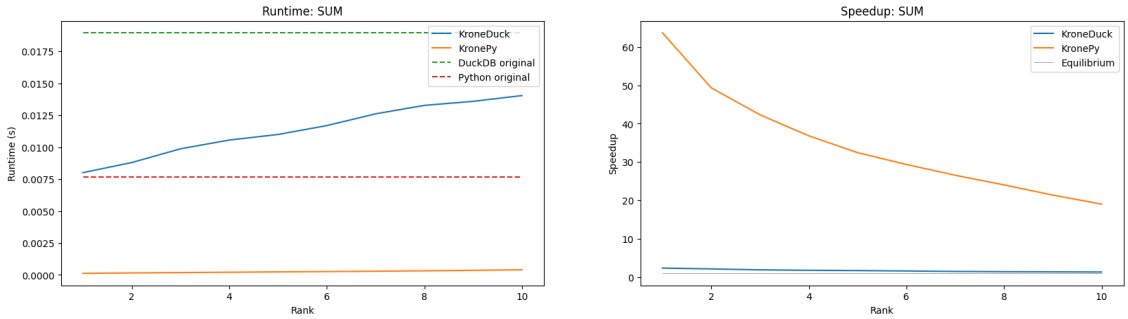


Figure 5.2: Runtime and speedup of a sum query on the value column.

In Figure 5.2, we observe that both KroneDuck and KronePy are significantly faster. For rank 1, KronePy is more than 63x faster and KroneDuck is more than 2.3x faster. The speedup is reduced for higher ranks until it is around 19x for KronePy and 1.3x for KroneDuck at rank 10.

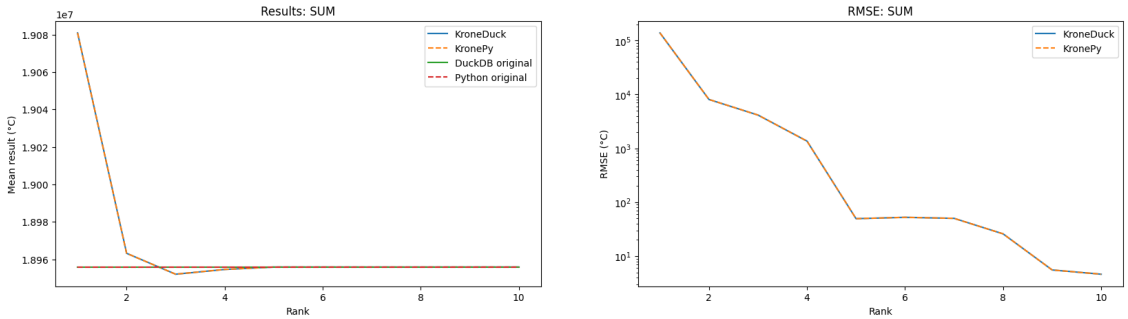


Figure 5.3: Mean result and root mean squared error (RMSE) of a sum query on the value column.

The accuracy of the sum results is shown in Figure 5.3. To calculate the error of the accuracy, a root mean squared error (RMSE) is used over the sums for each key. For rank 1, this is around 138′000°C which corresponds to a 0.7% deviation from the correct result. This error shrinks drastically for higher ranks and is around 50°C for rank 5, which should in most applications be insignificant. For rank 10 the error is around 5°C for a sum of more than 1.7 million tuples per key.

5.5.4 Count

The same dataset as for the selection queries is used for the count query. The speedup for KroneDuck is around 2.3x for all ranks, because the count query is rank-independent. For the NumPy implementation, there is no speedup and KronePy is actually 20% slower than the baseline Python implementation. This is a difference of 0.4 microseconds and is most likely due to one additional multiplication per key.

5.5.5 Average

The same dataset as for the selection queries is used for the average query. The picture is very similar to the sum query. The speedup is 32x-14x for KronePy and 2.4x-1.3x for KroneDuck for ranks 1-10. The error starts at 0.08°C or 0.8% for rank 1 and becomes insignificant for ranks 5 and higher. The resulting plots can be seen in Figures 11.7 and 11.8, in the appendix.

5.5.6 Sum Product

For the sum product experiments, we used a dataset containing the nine different measurement seen in Table 4.1 for the three cities `KLO, BAS, BER`. We ran 100 epochs and at each epoch, a random selection of measurements was chosen. The results are averaged over the epochs. To avoid one combination of columns dominating the results, the columns were normalized by dividing them by their max value.

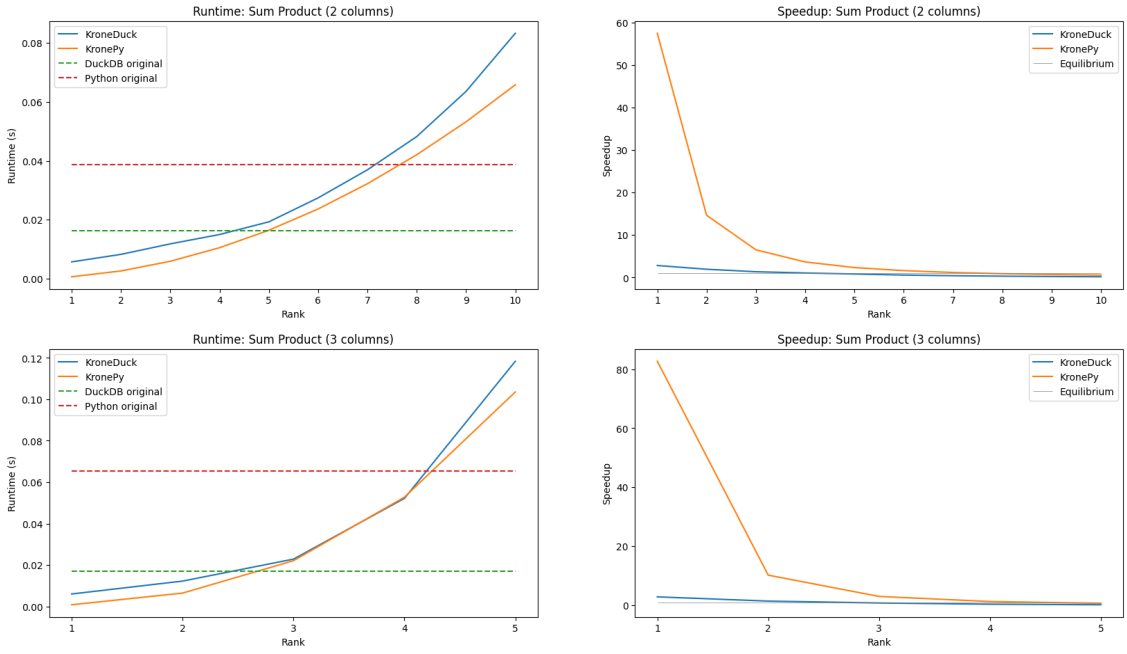


Figure 5.4: Runtime and speedup of a sum product query on the value column.

As seen in Figure 5.4, for the sum product of two columns, the runtime of the KroneDB implementations, which is quadratic in the rank, makes it that KroneDuck is only faster than DuckDB until rank 4 and KronePy is only faster until rank 7. Nevertheless, the advantages in the lower ranks are significant. KronePy is around 57x faster than the baseline Python implementation and KroneDuck is about 2.8x faster than DuckDB at rank 1. The speedup of KonePy is highly influenced because the baseline Python implementation is slower than DuckDB, which suggests that there is a lot of potential to improve the baseline, which would reduce the speedup. The

baseline Python implementation is not taking full advantage of hyperthreading and is therefore slower than DuckDB. More important is the speedup of KroneDuck, which is still 2x and 1.4x faster than DuckDB at rank 2 and 3 respectively.

Looking at the runtimes for the three-column sum product, the speedup of KroneDuck is still 2.8x and 1.4x at rank 1 and 2 respectively.

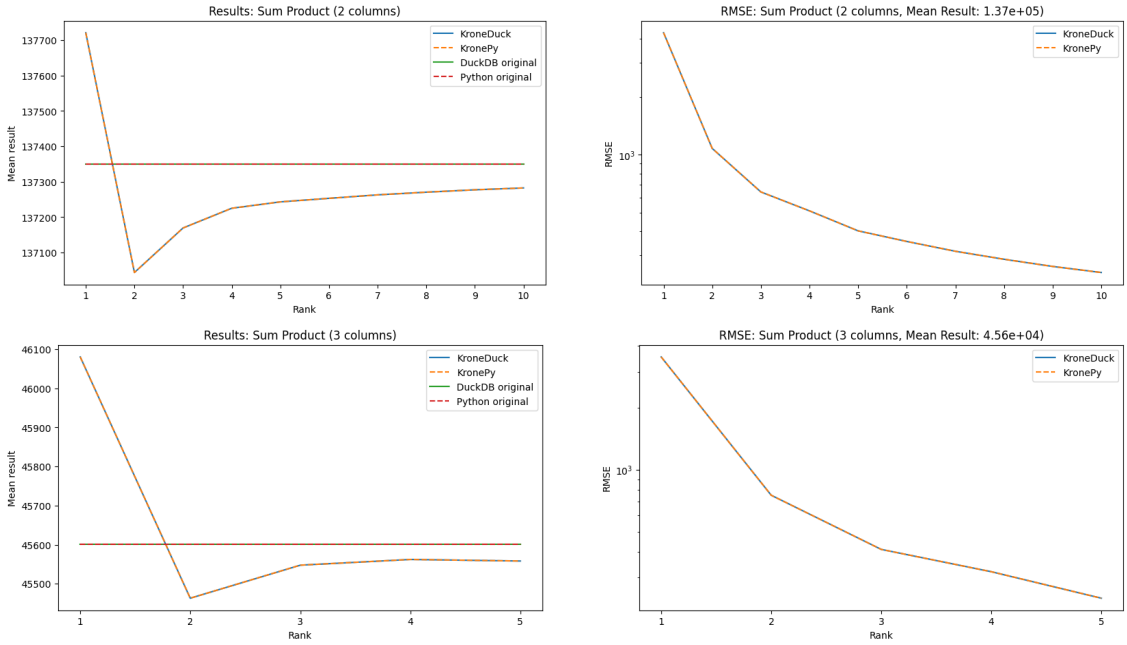


Figure 5.5: Mean result and root mean squared error (RMSE) of a sum product query on the value column.

For the two-column sum product, the accuracy of the results starts at an RMSE of 3% of the average result for rank 1 and drastically decreases to 0.8% and 0.5% for rank 2 and 3 respectively. This is shown in Figure 5.5, together with the results for the three-column sum product, where the RMSE starts at 7.8% for rank 1 and decreases to 1.6% for rank 2.

5.5.7 Min, Max

Because only rank 1 is considered, independent of the rank of the decomposition, the speedup is constant around 23x for KronePy and 2.2x for KroneDuck. The approximated minimum values have an RMSE of around 6.1°C and are all above the correct minimum values. This was expected because the approximation of the negative values is particularly bad as discussed in Section 4.4 and the approximation generally tends to be closer to the mean value than the min value. The approximated maximum values have a much smaller RMSE of around 1.3°C and are all below the correct maximum values for the same reason.

Chapter 6

Updates in KroneDB

In this chapter, we will discuss how to update decomposed KroneRelations. In Section 6.1, it is shown how new data can be added to a decomposed KroneRelation directly in the decomposed form. More advantages of this method are discussed in the Sections 6.2 and 6.3, where we show that it is trivial to impute missing values and even possible to detect anomalies in the new data, respectively. The experimental results are shown in Section 6.4.

6.1 Insert new Data

Consider the simple example

$$\underbrace{\begin{bmatrix} s_{11} \\ s_{21} \\ s_{31} \end{bmatrix}}_{\mathbf{s}_1} \otimes \underbrace{\begin{bmatrix} p_{11} \\ p_{21} \end{bmatrix}}_{\mathbf{p}_1} = \underbrace{\begin{bmatrix} s_{11}p_{11} \\ s_{11}p_{21} \\ \hline s_{21}p_{11} \\ s_{21}p_{21} \\ \hline s_{31}p_{11} \\ s_{31}p_{21} \end{bmatrix}}_{\mathbf{d}_{1,1}} = \underbrace{\begin{bmatrix} d_1 \\ d_2 \\ \hline d_3 \\ d_4 \\ \hline d_5 \\ d_6 \end{bmatrix}}_{\mathbf{d}_{1,1}},$$

where $\mathbf{s}_1$, $\mathbf{p}_1$ and $\mathbf{d}_{1,1}$ are the data-column vectors of $\mathbf{S}(X_1, RID, V_1)$, $\mathbf{P}(X_1, RID, P_1)$, and $\mathbf{D}(X_1, RID, D_1)$, respectively. The relation $\mathbf{D}$ is decomposed with a period of length 2 and is no longer in the database because only the decomposed relations are stored. Consider that $\mathbf{D}$ is updated with a new value d_7. The easiest way is to reconstruct the relation $\mathbf{D}$, add the new value d_7, and decompose it again. However, there are two major problems with this approach.

The most obvious problem is that the new relation $\mathbf{D}$ has seven tuples, which is a prime number, therefore it can only be decomposed with a period of length 1 or 7. Both cases are not optimal and use more space than just storing the original relation $\mathbf{D}$. It is however not common to get exactly a prime number of tuples by inserting a new tuple, but also a lot of non-prime numbers that are not optimal for decomposition.

The second problem is that reconstructing the relation $\mathbf{D}$ and decomposing it again is very expensive. Especially if the relation $\mathbf{D}$ is large, the decomposition using SVD is very time-consuming.

To address the first problem, suppose that the original choice of the period length m_p was with good reason. Therefore, it would be better to keep the period length even if new data is inserted. To do this, the new data needs to be collected in a buffer and inserted at once when the number of tuples in the buffer is equal to the period length m_p. By adding a complete period at once, the new relation $\mathbf{D}$ can be decomposed with the same period length as the original relation $\mathbf{D}$.

The second problem can be solved by using the fact that the new data is added to the end of the relation $\mathbf{D}$ and is, due to the introduced buffering, one whole new period. Looking at the

example above, adding another period would add the values d_7 and d_8, which need to be expressed as the period vector $\mathbf{p}_1$ multiplied by some scaling factor:

$$\underbrace{\begin{bmatrix} s_{11} \\ s_{21} \\ s_{31} \\ ? \end{bmatrix}}_{\mathbf{s}_1} \otimes \underbrace{\begin{bmatrix} p_{11} \\ p_{21} \end{bmatrix}}_{\mathbf{p}_1} = \underbrace{\begin{bmatrix} s_{11}p_{11} \\ s_{11}p_{21} \\ \hline s_{21}p_{11} \\ s_{21}p_{21} \\ \hline s_{31}p_{11} \\ s_{31}p_{21} \\ \hline ? \; p_{11} \\ ? \; p_{21} \end{bmatrix}}_{\mathbf{d}_{1,1}} = \underbrace{\begin{bmatrix} d_1 \\ d_2 \\ \hline d_3 \\ d_4 \\ \hline d_5 \\ d_6 \\ \hline d_7 \\ d_8 \end{bmatrix}}_{\mathbf{d}_{1,1}}.$$

This scaling factor can be freely chosen, to fit the new data into the period. There is no need to decompose the whole relation $\mathbf{D}$ again. Of course, this only provides a good approximation if we assume that the new period of data follows a similar pattern as the previous periods.

Finding the optimal scaling factor s_{41} is a linear least squares problem. Given the vector of the new data $\mathbf{d}_{new} = [d_7, d_8]^T$ and the period vector $\mathbf{p}_1 = [p_{11}, p_{21}]^T$, we want to find the scaling factor s_{41} that minimizes:

$$\|\mathbf{d}_{new} - s_{41}\mathbf{p}_1\|_2.$$

In the simple case, where we only have a single value column and a decomposition of rank 1, the solution is given by projecting the new data vector $\mathbf{d}_{new}$ onto the period vector $\mathbf{p}_1$:

$$s_{41} = \frac{\langle \mathbf{p}_1, \mathbf{d}_{new} \rangle}{\langle \mathbf{p}_1, \mathbf{p}_1 \rangle}. \tag{6.1}$$

If the construction of the Kronecker decomposition is done, such that the singular values are multiplied into the scaling factors, the solution is given by

$$s_{41} = \langle \mathbf{p}_1, \mathbf{d}_{new} \rangle, \tag{6.2}$$

because the period vector $\mathbf{p}_1$ is normalized.

If we introduce rank k, we get a new optimization problem, where we want to minimize

$$\|\mathbf{d}_{new} - (s_{41}^1\mathbf{p}_1^1 + s_{41}^2\mathbf{p}_1^2 + ... + s_{41}^k\mathbf{p}_1^k)\|_2,$$

where $\mathbf{p}_1^i$ is the i-th rank of the period vector $\mathbf{p}_1$ and s_{41}^i is the scaling factor for the i-th rank. This is again a linear least squares problem, where the ranks of the period vector $\mathbf{p}_1$ can be seen as the columns of a matrix P' and the scaling factors s_{41}^i as the rows of a vector $\mathbf{s}'$. The resulting optimization problem is minimizing

$$\|\mathbf{d}_{new} - P'\mathbf{s}'\|_2.$$

By construction of the Kronecker decomposition using SVD, we know that the ranks of the period vector $\mathbf{p}_1$ are orthogonal, therefore the columns of the matrix P' are orthogonal. This means, that we can solve the optimization problem by projecting the new data vector $\mathbf{d}_{new}$ onto the columns of the matrix P':

$$\mathbf{s}' = diag(P'^T P')^{-1} P'^T \mathbf{d}_{new},$$

where $diag(P'^T P')$ is a diagonal matrix with the squared norm of the columns of the matrix P' on the diagonal which is used to normalize the columns of the matrix P'. This corresponds to $\frac{1}{\langle \mathbf{p}_1, \mathbf{p}_1 \rangle}$ in (6.1).

The last thing to introduce is multiple value columns. In this case, we have a matrix P of period vectors and a matrix S of scaling factors. The optimization problem is minimizing

$$\|D_{new} - s_{41}P\|_F,$$

where D_{new} is a matrix of new data vectors and $\|\cdot\|_F$ is the Frobenius norm. The solution is given by stacking the columns of the matrix P into a vector $\mathbf{p}$ and the columns of the matrix D_{new} into a vector $\mathbf{d}_{new}$. This way, we successfully reduced the problem to the previous case, where we had a single value column.

The general solution corresponding to (6.1) is then given by

$$\mathbf{s}' = diag(P'^T P')^{-1} P'^T \mathbf{d}_{new},$$

where

$$P' = \begin{bmatrix} vec(P_1^1) & vec(P_1^2) & \dots & vec(P_1^k) \end{bmatrix} \text{ and } \mathbf{d}_{new} = vec(D_{new}).$$

If the period vectors P_1^i are normalized we can again leave out the normalization as in (6.2) and get:

$$\mathbf{s}' = P'^T \mathbf{d}_{new}.$$

6.2 Value Imputation

Consider a slightly different example:

$$\underbrace{\begin{bmatrix} s_{11} \\ s_{21} \\ ? \end{bmatrix}}_{\mathbf{s}_1} \otimes \underbrace{\begin{bmatrix} p_{11} \\ p_{21} \\ p_{31} \end{bmatrix}}_{\mathbf{p}_1} = \underbrace{\left[\begin{array}{c} s_{11}p_{11} \\ s_{11}p_{21} \\ s_{11}p_{31} \\ \hline s_{21}p_{11} \\ s_{21}p_{21} \\ s_{21}p_{31} \\ \hline ?\ p_{11} \\ ?\ p_{21} \\ ?\ p_{31} \end{array}\right]}_{\mathbf{d}_{1,1}} = \underbrace{\left[\begin{array}{c} d_1 \\ d_2 \\ d_3 \\ \hline d_4 \\ d_5 \\ d_6 \\ \hline d_7 \\ \texttt{NULL} \\ d_9 \end{array}\right]}_{\mathbf{d}_{1,1}}.$$

In this example, we have a missing value in the new data vector $\mathbf{d}_{new} = [d_7, \texttt{NULL}, d_9]^T$. The value imputation problem is to find a good approximation for the missing value `NULL`.

There are multiple ways to solve this problem, using a mean or median value, or using linear or polynomial interpolation would be the simplest approach. More sophisticated approaches could be regression models like linear regression or even a principal component analysis (PCA) [AW10]. The obvious parallels between the Kronecker decomposition or low-rank decomposition in general and PCA suggest that we do not need to use an additional regression model, but can use the Kronecker decomposition directly to solve the value imputation problem.

Using the same approach as in the previous section, we can find a scaling factor s_{31} that minimizes the error for the known values d_7 and d_9 by removing the missing value from the vector $\mathbf{d}_{new}$ and the period vector $\mathbf{p}_1$ to get $\mathbf{d}'_{new} = [d_7, d_9]^T$ and $\mathbf{p}'_1 = [p_{11}, p_{31}]^T$. The scaling factor s_{31} is then given by the projection of the known values $\mathbf{d}'_{new}$ onto the reduced period vector $\mathbf{p}'_1$:

$$s_{31} = \langle \mathbf{p}'_1, \mathbf{d}'_{new} \rangle.$$

The missing value `NULL` can then be approximated by $d_8 = s_{31}p_{21}$.

6.3 Detecting Anomalies

Looking at Figure 6.1, we can see how large the error for a new day can get, depending on the number of days we decompose initially. We could use this knowledge to detect if the error for a new day is too large and therefore the new day might contain erroneous measurements.

This could in a first step be used to inform the user that the new data might be erroneous and should be checked.

In a second step, we could use this information to automatically detect erroneous data and correct it using value imputation. A simple algorithm to correct erroneous data could be the following:

1. Fit the new data using the update function.
2. Check the error for the new data and if it surpasses a threshold, continue with the next step.
3. Loop:
   (a) Calculate the error for each value in the new data compared to the fitted data.
   (b) Replace the value with the highest error with `NULL`.
   (c) Fit the new data using value imputation and recalculate the error.
   (d) If the error is smaller than the threshold, stop.

6.4 Experiments

6.4.1 Accuracy of Updates

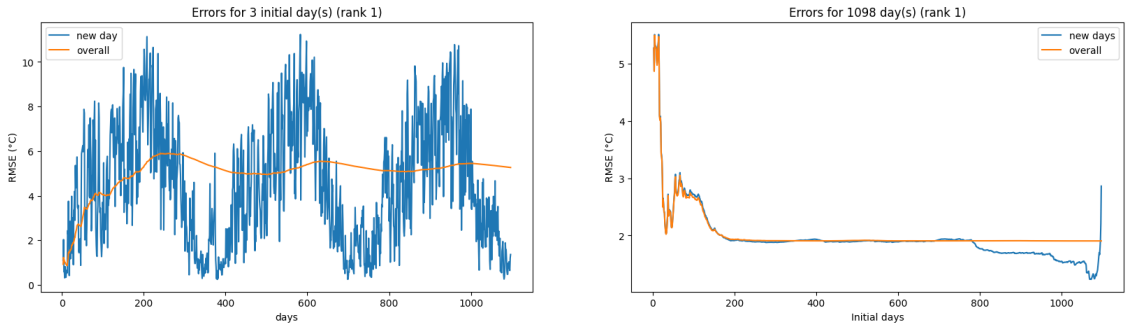


Figure 6.1: Insert a new day of temperature measurements in Kloten. The left plot shows the development of the error if we decompose the first three days and continue to add new days over three years. The right plot shows the development of the error if we decompose a specific number of days and then add the remaining days of the three years using updates.

In Figure 6.1 we can see the accuracy of the updates on the MeteoSwiss data set, specifically the temperature data in Kloten. The first plot shows the error if we only decompose three days and then add the remaining days one by one. In blue, we can see something that looks like the yearly pattern of the temperature seen in Figure 4.1. This pattern is the result of the first three days being winter days, which means that the decomposition is good for winter days, but not for other days. The more the new day differs from the first three days, the higher the error gets, which results in high errors for summer days and low errors for winter days. The approximation error over all days is shown in orange and suggests that the overall error stabilizes after about 200 days in between 5°C and 6°C for rank 1.

The second plot shows the error if we decompose a specific number of days and then add the remaining days of the three years using updates. The blue line shows the approximation error over all days that are added using updates. The overall error over all 1098 days is again shown in orange. We can see that the overall error stabilizes after about 200 days, which suggests that it could be enough to decompose the first 200 days and then add the remaining days using updates. The dip of the blue line towards the end is probably caused by the fact that the last 300 days are approximated exceptionally well because there are almost no winter days left, which are hard to approximate as we have seen in Section 4.4. The steep rise for the last few days is exactly because this is the start of the next winter.

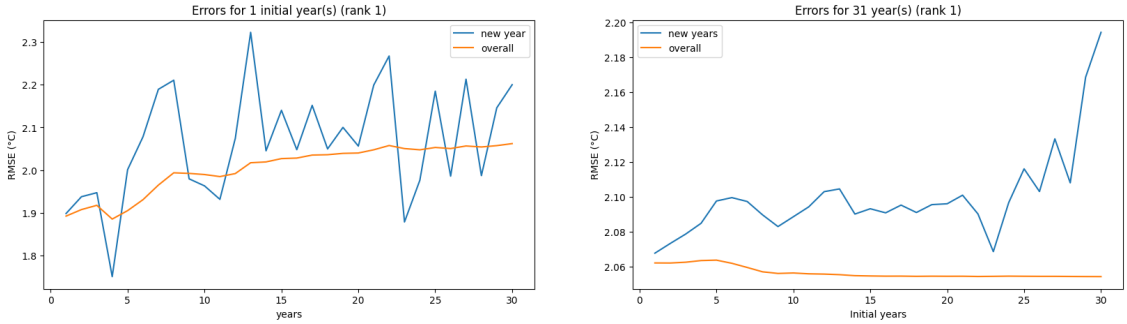


Figure 6.2: Insert a new year of temperature measurements in Kloten. The left plot shows the development of the error if we decompose the first year and continue to add new years over 31 years. The right plot shows the development of the error if we decompose a specific number of years and then add the remaining years of the 31 years using updates.

Figure 6.2 shows the same experiment as Figure 6.1 but for years instead of days. The first plot shows the error if we only decompose the first year and then add the remaining years one by one. Compared to Figure 6.1, the overall error stabilizes at a much lower value of about 2°C. This is because the initial decomposed year already contains all seasons and no completely unknown daily patterns are added. The error for the new year fluctuates in a comparably narrow range of about 1.7°C to 2.4°C.

The second plot again shows the error if we decompose a specific number of years and then add the remaining years of the 31 years using updates. The blue line shows the approximation error over all years that are added during the updates and suggests that at least the last two years are slightly different from the other years and are therefore approximated worse. Overall, however, it can be said that the error is very stable, especially after an initial nine years. It seems that the years six to nine add the necessary variety to the decomposition to approximate the remaining years slightly better than the first five years alone.

These experiments conclude that it is possible to decompose a small number of days, around 200, and then add the remaining days using updates without having a significant drop in reconstruction accuracy, because the most variation is captured. If the very small gain in accuracy is important, it is helpful to decompose multiple years, in this case, around nine, to capture the yearly variation well. This is of course only valid for this dataset and might be different for other datasets.

6.4.2 Accuracy of Value Imputation

In Figure 6.3 we can see the accuracy of the value imputation on the MeteoSwiss data set, specifically the temperature data in Kloten. The results suggest that using higher ranks for value imputation can hurt the accuracy if there are many missing values. This is probably because the higher ranks are more sensitive to outliers, which have a greater influence if many other values are

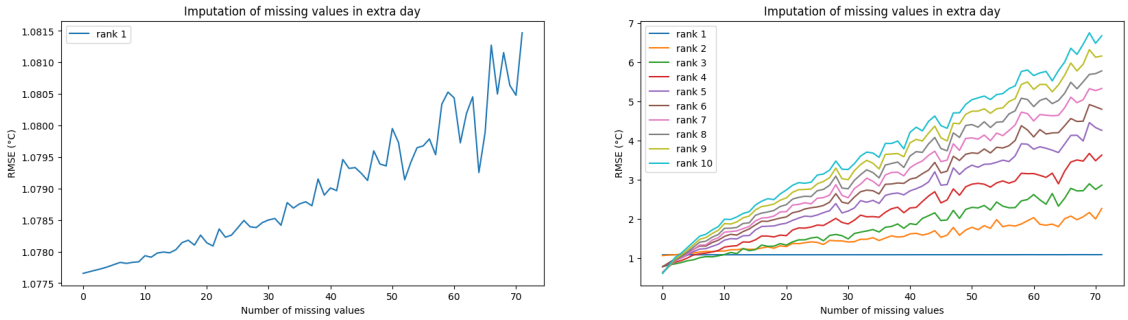


Figure 6.3: Value imputation on the MeteoSwiss data set, specifically the temperature data in Kloten. This is the result of decomposing the data over the whole period of 32 years and updating the relation with one new day. The new day has a fixed number of randomly missing values, which are then imputed using the method described in Section 6.2. The error is measured as the root mean square error (RMSE) over the new day.

missing. It might even be a problem that they lose the ability to influence the scaling if they are missing themselves, but they have a great impact on the root mean square error.

Therefore, if there are many missing values, it is probably better to do the value imputation and update in two steps. First, impute the missing values using a low rank and then update the relation with the imputed values using a higher rank decomposition.

For very few missing values, it is still significantly better to use the highest available rank for value imputation. The same caveat as in the previous section applies, that this is only valid for this dataset and must be tested for the specific dataset that is used.

Chapter 7

KroneDB in FAQ

This is a more formal and general treatment of Kronecker decomposition within the general framework of FAQ. It is discussed how KroneRelations can be expressed as FAQ factors and how these factors can be replaced by the corresponding decomposed relations, to get a new FAQ over the decomposed relations in Section 7.1. We provide specific examples of aggregations as FAQs and how they can be modified to use the decomposed relations in Section 7.2, Section 7.3, and Section 7.4. Finally, we discuss how a general FAQ containing KroneRelations can be rewritten and evaluated using the InsideOut algorithm in Section 7.5.

The conclusion of this chapter is a generally applicable method to rewrite FAQs containing KroneRelations such that they can be evaluated over the decomposed relations. We show that this evaluation can be done using the InsideOut algorithm without any modifications. Since FAQs are known to capture computation across many areas, including query evaluation in databases, constraint satisfaction problems, linear algebra (matrix chain multiplication, discrete Fourier transform), satisfiability, inference and learning in probabilistic graphical models, machine learning (cost functions, covariance matrices, learning), and tensor networks in physics. By showing how our work integrates naturally with FAQ, we effectively enable its use across all these fundamental problems.

7.1 KroneRelations as FAQ factors

Recall the representation of a Relation as a FAQ factor from Section 2.2. The FAQ factor for the data-relation $\mathbf{D}(\mathbf{x}, rid, \mathbf{d})$ is defined as

$$\psi_{\mathbf{x},rid,\mathbf{d}}(\mathbf{x}, rid, \mathbf{d}) = \begin{cases} \mathbf{1} & \text{if the tuple } (\mathbf{x}, rid, \mathbf{d}) \text{ is in } \mathbf{D}, \\ \mathbf{0} & \text{otherwise,} \end{cases}$$

where $\mathbf{x} = (x_1, \ldots, x_n)$ and $\mathbf{d} = (d_1, \ldots, d_{n_d}) = (d_{1,1}, \ldots, d_{n_s,n_p})$.

The FAQ factors for the scaling and period-relations $\mathbf{S}(\mathbf{x}, rid_s, \mathbf{s}, r)$ and $\mathbf{P}(\mathbf{x}, rid_p, \mathbf{p}, r)$ are defined as

$$\sigma(\mathbf{x}, rid_s, \mathbf{s}, r) = \psi_{\mathbf{x},rid_s,\mathbf{s},r}(\mathbf{x}, rid_s, \mathbf{s}, r) = \begin{cases} \mathbf{1} & \text{if the tuple } (\mathbf{x}, rid_s, \mathbf{s}, r) \text{ is in } \mathbf{S}, \\ \mathbf{0} & \text{otherwise,} \end{cases}$$

where $\mathbf{s} = (s_1, \ldots, s_{n_s})$ and

$$\rho(\mathbf{x}, rid_p, \mathbf{p}, r) = \psi_{\mathbf{x},rid_p,\mathbf{p},r}(\mathbf{x}, rid_p, \mathbf{p}, r) = \begin{cases} \mathbf{1} & \text{if the tuple } (\mathbf{x}, rid_p, \mathbf{p}, r) \text{ is in } \mathbf{P}, \\ \mathbf{0} & \text{otherwise,} \end{cases}$$

where $\mathbf{p} = (p_1, \ldots, p_{n_p})$ respectively.

Using the FAQ framework allows us to substitute a KroneRelation with the corresponding decomposed relations by replacing the factor $\psi_{\mathbf{x},rid,\mathbf{d}}(\mathbf{x}, rid, \mathbf{d})$ with the factors $\sigma(\mathbf{x}, rid_s, \mathbf{s}, r)$ and $\rho(\mathbf{x}, rid_p, \mathbf{p}, r)$ as follows:

$$\psi_{\mathbf{x},rid,\mathbf{d}}(\mathbf{x}, rid, \mathbf{d}) \approx \sum_{rid_s, rid_p, \mathbf{s}, \mathbf{p}, r} \sigma(\mathbf{x}, rid_s, \mathbf{s}, r) \cdot \rho(\mathbf{x}, rid_p, \mathbf{p}, r) \cdot (\mathbf{d} = \mathbf{s} * \mathbf{p}) \cdot (rid = (rid_s - 1) \cdot m_{p;\mathbf{x}} + rid_p). \tag{7.1}$$

The expressions $(\mathbf{d} = \mathbf{s} * \mathbf{p})$ and $(rid = (rid_s - 1) \cdot m_{p;\mathbf{x}} + rid_p)$ are virtual factors. These are not real relations but are used to define the mapping between the data-relation and the scaling and period-relations.

The first virtual factor $(\mathbf{d} = \mathbf{s} * \mathbf{p})$ is defined as

$$\psi_{\mathbf{d},\mathbf{s},\mathbf{p}}(\mathbf{d}, \mathbf{s}, \mathbf{p}) = \begin{cases} \mathbf{1} & \text{if } d_{u,v} = s_u \cdot p_v \text{ for all } u \in [n_p], v \in [n_s], \\ \mathbf{0} & \text{otherwise.} \end{cases} \tag{7.2}$$

and maps the value columns of the data-relation to the value columns of the scaling and period-relations according to (5.1).

The second virtual factor $(rid = (rid_s - 1) \cdot m_{p;\mathbf{x}} + rid_p)$ is defined as

$$\psi_{\mathbf{x},rid,rid_s,rid_p}(\mathbf{x}, rid, rid_s, rid_p) = \begin{cases} \mathbf{1} & \text{if } rid = (rid_s - 1) \cdot m_{p;\mathbf{x}} + rid_p, \\ \mathbf{0} & \text{otherwise,} \end{cases}$$

where $m_{p;\mathbf{x}}$ is the period length of the Kronecker decomposition for the key values $\mathbf{x}$. This factor is responsible for ensuring the correct mapping of the rows between the data-relation and the scaling and period-relations according to (4.1).

7.2 The Sum as FAQ

We will start with the sum over a single value column $d_{u,v}$ in the data-relation $\mathbf{D}$. This sum can be expressed as

$$\phi_{sum}(\mathbf{x}_f) = \sum_{\mathbf{x}_{f+1,\dots,n}, rid, \mathbf{d}} \psi_{\mathbf{x},rid,\mathbf{d}}(\mathbf{x}, rid, \mathbf{d}) \cdot d_{u,v}. \tag{7.3}$$

Recall form Section 2.2 that $\mathbf{x}_f$ is the set of free variables and $\mathbf{x}_{f+1,\dots,n} = x_{f+1}, \dots, x_n$ is the set of bound variables in $\mathbf{x}$ and that $d_{u,v}$ is the virtual factor defined as $\psi_{d_{u,v}}(d_{u,v}) = d_{u,v}$.

Using a rank-1 decomposition, we can rewrite the sum query by substituting the factor $\psi_{\mathbf{x},rid,\mathbf{d}}(\mathbf{x}, rid, \mathbf{d})$ with the decomposed factors according to (7.1) as follows:

$$\phi'_{sum}(\mathbf{x}_f) = \sum_{\mathbf{x}_{f+1,\dots,n}, rid_s, rid_p, \mathbf{s}, \mathbf{p}, \mathbf{d}} \sigma(\mathbf{x}, rid_s, \mathbf{s}) \cdot \rho(\mathbf{x}, rid_p, \mathbf{p}) \cdot d_{u,v} \cdot (\mathbf{d} = \mathbf{s} * \mathbf{p}) \tag{7.4}$$

$$= \sum_{\mathbf{x}_{f+1,\dots,n}, rid_s, rid_p, \mathbf{s}, \mathbf{p}} \sigma(\mathbf{x}, rid_s, \mathbf{s}) \cdot \rho(\mathbf{x}, rid_p, \mathbf{p}) \cdot s_u \cdot p_v \tag{7.5}$$

$$= \sum_{\mathbf{x}_{f+1,\dots,n}} \left(\sum_{rid_s, \mathbf{s}} \sigma(\mathbf{x}, rid_s, \mathbf{s}) \cdot s_u \right) \cdot \left(\sum_{rid_p, \mathbf{p}} \rho(\mathbf{x}, rid_p, \mathbf{p}) \cdot p_v \right). \tag{7.6}$$

The rank variable r is omitted because we are using a rank-1 decomposition.

The Equation 7.4 is the result of substituting the original relation in (7.3) with the decomposed relations defined in (7.1). The virtual factor $(rid = (rid_s - 1) \cdot m_{p;\mathbf{x}} + rid_p)$ can be omitted, because rid is uniquely defined by rid_s and rid_p and is aggregated over. This allows rid to be removed from the FAQ completely and makes the values rid_s and rid_p independent of each other.

Equation 7.5 is the result of substituting the variables in $\mathbf{d}$ with the corresponding variables in $\mathbf{s}$ and $\mathbf{p}$ according to the column mapping in (7.2). The only variable that is relevant for the sum is $d_{u,v}$, which is substituted with $s_u \cdot p_v$. After the substitution, the variable $\mathbf{d}$ and the virtual factor ($\mathbf{d} = \mathbf{s} * \mathbf{p}$) can be removed from the FAQ, because $\mathbf{d}$ is uniquely defined by $\mathbf{s}$ and $\mathbf{p}$ and is aggregated over.

From (7.5) to (7.6) we push the aggregation over rid_s and s_u down to $\sigma(\mathbf{x}, rid_s, \mathbf{s})$ and the aggregation over rid_p and p_v to $\rho(\mathbf{x}, rid_p, \mathbf{p})$, because those aggregations are independent of each other. Therefore, we get two independent sums over the scaling and period-relation, which can be calculated individually. Comparing this to (5.2), we can see that we reached the same conclusion by rewriting the sum in FAQ and linear algebra.

For the rank-k decomposition, we no longer omit the rank variable r and get the following FAQ:

$$\phi'_{sum}(\mathbf{x}_f) = \sum_{\mathbf{x}_{f+1,\dots,n},r} \left(\sum_{rid_s,\mathbf{s}} \sigma(\mathbf{x}, rid_s, \mathbf{s}, r) \cdot s_u \right) \cdot \left(\sum_{rid_p,\mathbf{p}} \rho(\mathbf{x}, rid_p, \mathbf{p}, r) \cdot p_v \right).$$

This is again parallel to the linear algebra representation in (5.3).

7.3 Other Aggregates as FAQs

This section briefly discusses how the other aggregates can be expressed in FAQ.

7.3.1 The Count as FAQ

The count as FAQ over the data-relation $\mathbf{D}(\mathbf{x}, rid, \mathbf{d})$ is defined as

$$\phi_{count}(\mathbf{x}_f) = \sum_{\mathbf{x}_{f+1,\dots,n},rid,\mathbf{d}} \psi_{\mathbf{x},rid,\mathbf{d}}(\mathbf{x}, rid, \mathbf{d}).$$

Using the decomposed relations $\mathbf{S}(\mathbf{x}, rid_s, \mathbf{s}, r)$ and $\mathbf{P}(\mathbf{x}, rid_p, \mathbf{p}, r)$, the count can be expressed as

$$\begin{aligned}\phi'_{count}(\mathbf{x}_f) &= \sum_{\mathbf{x}_{f+1,\dots,n},rid_s,rid_p,\mathbf{s},\mathbf{p},r} \sigma(\mathbf{x}, rid_s, \mathbf{s}, r) \cdot \rho(\mathbf{x}, rid_p, \mathbf{p}, r) \cdot (r = 1) \\ &= \sum_{\mathbf{x}_{f+1,\dots,n}} \left(\sum_{rid_s,\mathbf{s},r} \sigma(\mathbf{x}, rid_s, \mathbf{s}, r) \cdot (r = 1) \right) \cdot \left(\sum_{rid_p,\mathbf{p},r} \rho(\mathbf{x}, rid_p, \mathbf{p}, r) \cdot (r = 1) \right).\end{aligned}$$

7.3.2 The Average as FAQ

The average is simply the sum divided by the count, therefore can the sum and the count query be used to calculate the average. It is not possible to express the average as a single FAQ but the sum and count can be expressed as a single FAQ by using the sum-product semiring over pairs $((\mathbb{R}, \mathbb{R}), (+,+), (\cdot,\cdot), (0,0), (1,1))$. Thus,

$$\phi_{(sum,count)}(\mathbf{x}_f) = \sum_{\mathbf{x}_{f+1,\dots,n},rid,\mathbf{d}} \psi_{\mathbf{x},rid,\mathbf{d}}(\mathbf{x}, rid, \mathbf{d}) \cdot (d_{u,v}, 1) \ ,$$

where $\psi_{d_{u,v}}(d_{u,v}) = (d_{u,v}, 1)$.

The sum-count FAQ over the decomposed relations is defined as

$$\phi'_{(sum,count)}(\mathbf{x}_f) =$$

$$\sum_{\mathbf{x}_{f+1,\dots,n},r} \left(\sum_{rid_s,\mathbf{s}} \sigma(\mathbf{x}, rid_s, \mathbf{s}, r) \cdot (s_u, 1) \cdot (r = 1) \right) \cdot \left(\sum_{rid_p,\mathbf{p}} \rho(\mathbf{x}, rid_p, \mathbf{p}, r) \cdot (p_v, 1) \cdot (r = 1) \right) \ ,$$

where $\psi_{s_u}(s_u) = (s_u, 1)$, $\psi_{p_v}(p_v) = (p_v, 1)$ and

$$(r = 1) = \psi_r(r) = \begin{cases} (1,1) & \text{if } r = 1, \\ (1,0) & \text{otherwise.} \end{cases}$$

7.3.3 The Minimum and Maximum as FAQs

The minimum and maximum queries can be expressed as FAQ by using the min-product and max-product semirings respectively. The min-product semiring is defined as $(\mathbb{R}, \min, \cdot, \infty, 0)$ and the max-product semiring is defined as $(\mathbb{R}, \max, \cdot, -\infty, 0)$. The FAQ for the minimum is defined as

$$\phi_{min}(\mathbf{x}_f) = \min_{\mathbf{x}_{f+1,\dots,n}, rid, \mathbf{d}} \quad \psi_{\mathbf{x}, rid, \mathbf{d}}(\mathbf{x}, rid, \mathbf{d}) \cdot d_{u,v}.$$

Substituting the data-relation with the decomposed relations results in the following FAQ:

$$\begin{aligned} \phi'_{min}(\mathbf{x}_f) &= \min_{\mathbf{x}_{f+1,\dots,n}, rid_s, rid_p, \mathbf{s}, \mathbf{p}, \mathbf{d}, r} \quad \sigma(\mathbf{x}, rid_s, \mathbf{s}, r) \cdot \rho(\mathbf{x}, rid_p, \mathbf{p}, r) \cdot d_{u,v} \cdot (\mathbf{d} = \mathbf{s} * \mathbf{p}) \cdot (r = 1) \\ &= \min_{\mathbf{x}_{f+1,\dots,n}, rid_s, rid_p, \mathbf{s}, \mathbf{p}, r} \quad \sigma(\mathbf{x}, rid_s, \mathbf{s}, r) \cdot \rho(\mathbf{x}, rid_p, \mathbf{p}, r) \cdot s_u \cdot p_v \cdot (r = 1). \end{aligned}$$

The FAQ for the maximum query is defined analogously.

7.4 The Sum Product as FAQ

As discussed in Section 5.3.5, the sum product is a general case of the sum in linear algebra. The same is true for FAQ, where the sum is the sum product over a single value column and the sum product is expressed like the sum but with multiple virtual factors for the value columns. Thus,

$$\phi_{sp}(\mathbf{x}_f) = \sum_{\mathbf{x}_{f+1,\dots n}, rid, \mathbf{d}} \quad \psi_{\mathbf{x}, rid, \mathbf{d}}(\mathbf{x}, rid, \mathbf{d}) \cdot d_{u_1,v_1} \cdot \ldots \cdot d_{u_l,v_l},$$

where $\{d_{u_1,v_1}, \dots, d_{u_l,v_l}\} = \mathbf{d}_l \subseteq \mathbf{d}$ is the subset of variables which are used in the sum product.

As seen for the other aggregates, the factor for the data-relation $\psi_{\mathbf{x}, rid, \mathbf{d}}(\mathbf{x}, rid, \mathbf{d})$ can be replaced by the decomposed factors to get a new FAQ over the decomposed relations. For rank 1 this is trivial and we get

$$\begin{aligned} &\phi'_{sp}(\mathbf{x}_f) \\ &= \sum_{\mathbf{x}_{f+1,\dots,n}, rid_s, rid_p, \mathbf{s}, \mathbf{p}, \mathbf{d}} \quad \sigma(\mathbf{x}, rid_s, \mathbf{s}) \cdot \rho(\mathbf{x}, rid_p, \mathbf{p}) \cdot d_{u_1,v_1} \cdot \ldots \cdot d_{u_l,v_l} \cdot (\mathbf{d} = \mathbf{s} * \mathbf{p}) \\ &= \sum_{\mathbf{x}_{f+1,\dots,n}, rid_s, rid_p, \mathbf{s}, \mathbf{p}} \quad \sigma(\mathbf{x}, rid_s, \mathbf{s}) \cdot \rho(\mathbf{x}, rid_p, \mathbf{p}) \cdot s_{u_1} \cdot p_{v_1} \cdot \ldots \cdot s_{u_l} \cdot p_{v_l} \\ &= \sum_{\mathbf{x}_{f+1,\dots,n}} \left(\sum_{rid_s, \mathbf{s}} \quad \sigma(\mathbf{x}, rid_s, \mathbf{s}) \cdot s_{u_1} \cdot \ldots \cdot s_{u_l} \right) \cdot \left(\sum_{rid_p, \mathbf{p}} \quad \rho(\mathbf{x}, rid_p, \mathbf{p}) \cdot p_{v_1} \cdot \ldots \cdot p_{v_l} \right), \end{aligned}$$

where the rank variable r is omitted because we are using a rank-1 decomposition. This is the same conclusion as was reached in (5.8).

For the rank-k decomposition, we need to make sure that the summation over the rank variable r is done before the multiplication over the value columns as was done in (5.10). To ensure this, we need to rewrite the query over the data-relation to separate the virtual factors for the value columns and pair each value column with a separate data-relation factor. To ensure the elementwise multiplication over the value columns, they all share the same *rid*-value. This rewriting looks like

this:

$$\phi_{sp}(\mathbf{x}_f) = \sum_{\mathbf{x}_{f+1,\ldots,n},rid,\mathbf{d}} \psi_{\mathbf{x},rid,\mathbf{d}}(\mathbf{x},rid,\mathbf{d}) \cdot d_{u_1,v_1} \cdot \ldots \cdot d_{u_l,v_l}$$
$$= \sum_{\mathbf{x}_{f+1,\ldots,n},rid,\mathbf{d}_1,\ldots,\mathbf{d}_l} \psi_{\mathbf{x},rid,\mathbf{d}^1}(\mathbf{x},rid,\mathbf{d}^1) \cdot d^1_{u_1,v_1} \cdot \ldots \cdot \psi_{\mathbf{x},rid,\mathbf{d}^l}(\mathbf{x},rid,\mathbf{d}^l) \cdot d^l_{u_l,v_l}.$$

Each factor can then be replaced separately by the decomposed factors, to get:

$$\phi'_{sp}(\mathbf{x}_f) =$$

$$= \sum_{\mathbf{x}_{f+1,\ldots,n},rid,\mathbf{d}_1,\ldots,\mathbf{d}_l} \left(\sum_{rid^1_s,rid^1_p,\mathbf{s}^1,\mathbf{p}^1,r^1} \sigma(\mathbf{x},rid^1_s,\mathbf{s}^1,r^1) \cdot \rho(\mathbf{x},rid^1_p,\mathbf{p}^1,r^1) \cdot d^1_{u_1,v_1} \cdot (\mathbf{d}^1 = \mathbf{s}^1 * \mathbf{p}^1) \cdot (rid = (rid^1_s - 1) \cdot m_{p;\mathbf{x}} + rid^1_p) \right) \cdot \ldots$$
$$\cdot \left(\sum_{rid^l_s,rid^l_p,\mathbf{s}^l,\mathbf{p}^l,r^l} \sigma(\mathbf{x},rid^l_s,\mathbf{s}^l,r^l) \cdot \rho(\mathbf{x},rid^l_p,\mathbf{p}^l,r^l) \cdot d^l_{u_l,v_l} \cdot (\mathbf{d}^l = \mathbf{s}^l * \mathbf{p}^l) \cdot (rid = (rid^l_s - 1) \cdot m_{p;\mathbf{x}} + rid^l_p) \right) \tag{7.7}$$

$$= \sum_{\mathbf{x}_{f+1,\ldots,n},rid,rid_s,rid_p,\mathbf{d}^1,\mathbf{s}^1,\mathbf{p}^1,r^1,\ldots,\mathbf{d}^l,\mathbf{s}^l,\mathbf{p}^l,r^l} \sigma(\mathbf{x},rid_s,\mathbf{s}^1,r^1) \cdot \rho(\mathbf{x},rid_p,\mathbf{p}^1,r^1) \cdot d^1_{u_1,v_1} \cdot \ldots \cdot \sigma(\mathbf{x},rid_s,\mathbf{s}^l,r^l) \cdot \rho(\mathbf{x},rid_p,\mathbf{p}^l,r^l) \cdot d^l_{u_l,v_l}$$
$$\cdot (\mathbf{d}^1 = \mathbf{s}^1 * \mathbf{p}^1) \cdot \ldots \cdot (\mathbf{d}^l = \mathbf{s}^l * \mathbf{p}^l) \cdot (rid = (rid_s - 1) \cdot m_{p;\mathbf{x}} + rid_p) \tag{7.8}$$

$$= \sum_{\mathbf{x}_{f+1,\ldots,n},rid_s,rid_p,\mathbf{s}^1,\mathbf{p}^1,r^1,\ldots,\mathbf{s}^l,\mathbf{p}^l,r^l} \sigma(\mathbf{x},rid_s,\mathbf{s}^1,r^1) \cdot s^1_{u_1} \cdot \rho(\mathbf{x},rid_p,\mathbf{p}^1,r^1) \cdot p^1_{v_1} \cdot \ldots \cdot \sigma(\mathbf{x},rid_s,\mathbf{s}^l,r^l) \cdot s^l_{u_l} \cdot \rho(\mathbf{x},rid_p,\mathbf{p}^l,r^l) \cdot p^l_{v_l} \tag{7.9}$$

$$= \sum_{\mathbf{x}_{f+1,\ldots,n},r^1,\cdots,r^l} \sum_{rid_s} \left(\sum_{\mathbf{s}^1} \sigma(\mathbf{x},rid_s,\mathbf{s}^1,r^1) \cdot s^1_{u_1} \cdot \ldots \cdot \sum_{\mathbf{s}^l} \sigma(\mathbf{x},rid_s,\mathbf{s}^l,r^l) \cdot s^l_{u_l} \right)$$
$$\cdot \sum_{rid_p} \left(\sum_{\mathbf{p}^1} \rho(\mathbf{x},rid_p,\mathbf{p}^1,r^1) \cdot p^1_{v_1} \cdot \ldots \cdot \sum_{\mathbf{p}^l} \rho(\mathbf{x},rid_p,\mathbf{p}^l,r^l) \cdot p^l_{v_l} \right). \tag{7.10}$$

The equation (7.7) is the result of substituting each of the factors $\psi_{\mathbf{x},rid,\mathbf{d}^1}(\mathbf{x},rid,\mathbf{d}^1)$, ..., $\psi_{\mathbf{x},rid,\mathbf{d}^l}(\mathbf{x},rid,\mathbf{d}^l)$ with the corresponding decomposed factors according to (7.1).

Because rid and $\mathbf{x}$ uniquely define rid_s^i and rid_p^i for all $i \in [l]$, rid_s^i and rid_p^i respectively must be equal for all $i \in [l]$ and can be replaced with a single rid_s and rid_p. This simplification together with the pulling up of all aggregations is shown in (7.8).

From (7.8) to (7.9) we substitute the variables in $\mathbf{d}^1, \ldots, \mathbf{d}^l$ with the corresponding variables in $\mathbf{s}^1, \mathbf{p}^1, \ldots, \mathbf{s}^l, \mathbf{p}^l$ according to the column mapping $\mathbf{d}^i = \mathbf{s}^i * \mathbf{p}^i$ and can remove the column mapping factors and the variables $\mathbf{d}^1, \ldots, \mathbf{d}^l$ from the FAQ, as they are uniquely defined by $\mathbf{s}^1, \mathbf{p}^1, \ldots, \mathbf{s}^l, \mathbf{p}^l$ and are aggregated over. Additionally, can the rid variable together with the row mapping factor $(rid = (rid_s - 1) \cdot m_{p;\mathbf{x}} + rid_p)$ be removed, because rid is uniquely defined by rid_s, rid_p and $\mathbf{x}$ and is aggregated over.

In (7.10), the marginalization over rid_s, $\mathbf{s}^1, \ldots, \mathbf{s}^l$ and rid_p, $\mathbf{p}^1, \ldots, \mathbf{p}^l$ is pushed down to the corresponding factors. Comparing (7.10) to (5.12), shows that the conclusion is the same for FAQ and linear algebra.

7.5 Solving FAQs: The InsideOut Algorithm

A general algorithm to solve FAQ expressions is the InsideOut algorithm first introduced by Abo Khamis et al. [AKNR16]. In our description of the algorithm we closely follow the description in [Olt23]. The algorithm consists of two basic steps.

Step one in the *Marginalization* or *variable elimination* step, where bound variables are eliminated by marginalizing over them. Which bound variables are eliminated first is determined by a given marginalization order τ over the bound variables. Each marginalization step removes the rightmost bound variable x_i from τ and replaces the factors that contain x_i with a new factor that no longer contains x_i. To compute this new factor a worst-case optimal join algorithm like the Leapfrog Triejoin [Vel14] is used. This results in a simplified query with a hypergraph $\mathcal{H}'$ with fewer hyperedges and variables.

The second step is the *Evaluation* step. First, a hypertree decomposition $\mathcal{T}$ for the simplified hypergraph $\mathcal{H}'$ is constructed. The bags of the hypertree are materialized using Leapfrog Triejoin. Then, the new α-acyclic FAQ ϕ' is evaluated with Yannakakis' algorithm [Yan81].

7.5.1 Substituting KroneFactors

We describe how to use the InsideOut algorithm to evaluate FAQs in KroneDB where some of the factors are KroneRelations. For simplicity, we only consider FAQs over the sum-product semiring $(\mathbb{R}, +, \cdot, 0, 1)$, yet our approach can be generalized to arbitrary semi-rings. Assume we have a factor $\psi_K(\mathbf{x}_{K'}, rid, \mathbf{d})$ that corresponds to a KroneRelation with the corresponding hyperedge $K \in \mathcal{E}$, which we call a KroneFactor. Consider an FAQ where one of the factors is a KroneFactor:

$$\phi(\mathbf{x}_f) = \sum_{x_{f+1}} \cdots \sum_{x_n} \sum_{rid} \sum_{\mathbf{d}} \left(\prod_{S \in \mathcal{E} \setminus \{K\}} \psi_S(\mathbf{x}_S) \right) \cdot \psi_K(\mathbf{x}_{K'}, rid, \mathbf{d}) \cdot d_{u_1,v_1} \cdot \ldots d_{u_l,v_l}.$$

The hyperedge K has a corresponding KroneFactor $\psi_{K'}(\mathbf{x}_{K'}, rid, \mathbf{d}) = \psi_K(\mathbf{x}_K)$, where $K' \subset K$, such that $\mathbf{x}_K = \mathbf{x}_{K'} \cup \{rid\} \cup \mathbf{d}$.

- $\mathbf{x}_{K'} = \mathbf{x}_K \setminus (\{rid\} \cup \mathbf{d})$ is the set of variables that correspond to the key columns of the KroneRelation,
- rid, for which $\{rid\} = \mathbf{x}_K \setminus (\mathbf{x}_{K'} \cup \mathbf{d})$, is the variable that corresponds to the index column of the KroneRelation,
- $\mathbf{d} = \mathbf{x}_K \setminus (\mathbf{x}_{K'} \cup \{rid\})$ is the set of variables that correspond to the data columns of the KroneRelation,
- $d_{u_i,v_i} \in \mathbf{d}$ is the variable that corresponds to the data column of the KroneRelation that is used in the factor with the same name d_{u_i,v_i}.

Using the factor substitution rule from (7.9) we can rewrite the FAQ ϕ to

$$\phi'(\mathbf{x}_f) = \sum_{x_{f+1}} \cdots \sum_{x_n} \sum_{r^1,\ldots,r^l} \sum_{rid_s} \sum_{rid_p} \sum_{\mathbf{p}^1,\ldots,\mathbf{p}^l} \sum_{\mathbf{s}^1,\ldots,\mathbf{s}^l} \left(\prod_{S \in \mathcal{E} \setminus \{K\}} \psi_S(\mathbf{x}_S) \right)$$
$$\cdot \sigma(\mathbf{x}_{K'}, rid_s, \mathbf{s}^1, r^1) \cdot s^1_{u_1} \cdot \rho(\mathbf{x}_{K'}, rid_p, \mathbf{p}^1, r^1) \cdot p^1_{v_1} \cdot \ldots \cdot \sigma(\mathbf{x}_{K'}, rid_s, \mathbf{s}^l, r^l) \cdot s^l_{u_l} \cdot \rho(\mathbf{x}_{K'}, rid_p, \mathbf{p}^l, r^l) \cdot p^l_{v_l}.$$

This results in a new hypergraph $\mathcal{H}' = (\mathcal{V}', \mathcal{E}')$ with the set of vertices $\mathcal{V}'$ which is the set $\mathcal{V}$ without the indices for rid and $\mathbf{d}$ but with added indices for the new variables $r^1, \ldots, r^l$, rid_s, rid_p, $\mathbf{p}^1, \ldots, \mathbf{p}^l$ and $\mathbf{s}^1, \ldots, \mathbf{s}^l$. The new set of hyperedges is such that

$$\mathcal{E}' = \mathcal{E} \setminus \{K\} \cup \{\Sigma^1, \ldots, \Sigma^l, P^1, \ldots, P^l, S^1_{u_1}, \ldots, S^l_{u_l}, P^1_{v_1}, \ldots, P^l_{v_l}\}.$$

The hyperedges Σ^i and P^i correspond to the factors $\sigma(\mathbf{x}_{K'}, rid_s, \mathbf{s}^i, r^i)$ and $\rho(\mathbf{x}_{K'}, rid_p, \mathbf{p}^i, r^i)$, respectively and $S^i_{u_i}$ and $P^i_{v_i}$ are the hyperedges that correspond to the factors $s^i_{u_i}$ and $p^i_{v_i}$, respectively.

Marginalization Order

The choice of the marginalization order τ influences the runtime of the algorithm. There can be multiple optimal marginalization orders for a given FAQ. Given an optimal marginalization order τ for ϕ, by definition of the KroneRelation, we know that only the variables $\mathbf{x}_{K'}$ can occur in any other factor of ϕ, we can therefore assume that rid and $\mathbf{d}$ are the rightmost variables in τ without loss of optimality. The evaluation of ϕ' using InsideOut requires a new marginalization order τ', where the rightmost variables rid and $\mathbf{d}$ are removed from τ and replaced with the new variables $r^1, \ldots, r^l$, rid_s, rid_p, $\mathbf{s}^1, \ldots, \mathbf{s}^l$ and $\mathbf{p}^1, \ldots, \mathbf{p}^l$ in this order.

7.5.2 Evaluating ϕ'

If ϕ contains more than one KroneFactor, we can repeat the construction of ϕ', $\mathcal{H}'$ and τ' iteratively until all KroneFactors are substituted. Given the final ϕ' with the hypergraph $\mathcal{H}' = (\mathcal{V}', \mathcal{E}')$ and the marginalization order τ', it is possible to evaluate ϕ' using the InsideOut algorithm as described in [AKNR16] and [Olt23].

Semiring Marginalization

Recall that we only consider the sum-product semiring $(\mathbb{R}, +, \cdot, 0, 1)$ and that all marginalizations in ϕ' (and ϕ) use the sum. Therefore, each marginalization step of InsideOut applies Semiring Marginalization. To demonstrate how Semiring Marginalization works on a KroneFactor, the marginalization of $r^1, \ldots, r^l, rid_s, rid_p, \mathbf{s}^1, \ldots, \mathbf{s}^l, \mathbf{p}^1, \ldots, \mathbf{p}^l$ in ϕ' is explained on a high level. We will omit the effect of indicator projections at each step for simplicity. For this purpose, we only consider the factors that contain these bound variables. For demonstration purposes, a notation is used where the sums are already pushed down as in (7.10):

$$\begin{aligned}\phi'(\mathbf{x}_f) \quad = \cdots \sum_{r^1} \cdots \sum_{r^l} \quad & \sum_{rid_s} \left(\sum_{\mathbf{s}^1} \sigma(\mathbf{x}_{K'}, rid_s, \mathbf{s}^1, r^1) \cdot s^1_{u_1} \cdot \ldots \cdot \sum_{\mathbf{s}^l} \sigma(\mathbf{x}_{K'}, rid_s, \mathbf{s}^l, r^l) \cdot s^l_{u_l} \right) \\ & \cdot \sum_{rid_p} \left(\sum_{\mathbf{p}^1} \rho(\mathbf{x}_{K'}, rid_p, \mathbf{p}^1, r^1) \cdot p^1_{v_1} \cdot \ldots \cdot \sum_{\mathbf{p}^l} \rho(\mathbf{x}_{K'}, rid_p, \mathbf{p}^l, r^l) \cdot p^l_{v_l} \right).\end{aligned}$$

The marginalization over $\mathbf{p}^l$ removes the factors $\rho(\mathbf{x}_{K'}, rid_p, \mathbf{p}^l, r^l)$ and $p^l_{v_l}$ from ϕ' and replaces them with a new factor $\rho'(\mathbf{x}_{K'}, rid_p, r^l)$ in $|\mathbf{p}^l| = n_p$ steps. The new factor $\rho'(\mathbf{x}_{K'}, rid_p, r^l)$ returns the value for the attribute $p^l_{v_l}$ of the period-relation $\mathbf{P}$, for the unique tuple with the values

$\mathbf{x}_{K'}, rid_p$ and r^l if such a tuple exists, otherwise it returns $\mathbf{0}$. These marginalization steps are repeated for $\mathbf{p}^{l-1}, \ldots, \mathbf{p}^1$, which results in a new factors $\rho'(\mathbf{x}_{K'}, rid_p, r^{l-1}), \ldots, \rho'(\mathbf{x}_{K'}, rid_p, r^1)$. The same steps are repeated for $\mathbf{s}^l, \ldots, \mathbf{s}^1$, which results in a new factors $\sigma'(\mathbf{x}_{K'}, rid_s, r^l), \ldots, \sigma'(\mathbf{x}_{K'}, rid_s, r^1)$:

$$\begin{aligned}\phi''(\mathbf{x}_f) \qquad = \cdots \sum_{r^1} \cdots \sum_{r^l} \qquad & \left(\sum_{rid_s} \sigma'(\mathbf{x}_{K'}, rid_s, r^1) \cdot \ldots \cdot \sigma'(\mathbf{x}_{K'}, rid_s, r^l) \right) \\ & \cdot \left(\sum_{rid_p} \rho'(\mathbf{x}_{K'}, rid_p, r^1) \cdot \ldots \cdot \rho'(\mathbf{x}_{K'}, rid_p, r^l) \right).\end{aligned}$$

The next step is to marginalize over rid_p. This removes the factors $\rho'(\mathbf{x}_{K'}, rid_p, r^l), \ldots, \rho'(\mathbf{x}_{K'}, rid_p, r^1)$ and replaces them with a new factor $\rho''(\mathbf{x}_{K'}, r^1, \ldots, r^l)$. The new factor ρ'' returns the sum product of the columns $p_{v_1}, \ldots, p_{v_l}$ of the period-relation $\mathbf{P}$ where the rank attribute r is equal to $r^1, \ldots, r^l$, respectively. The same steps are repeated for rid_s, which results in a new factor $\sigma''(\mathbf{x}_{K'}, r^1, \ldots, r^l)$:

$$\phi'''(\mathbf{x}_f) \qquad = \cdots \sum_{r^1} \cdots \sum_{r^l} \left(\sigma''(\mathbf{x}_{K'}, r^1, \ldots, r^l) \cdot \rho''(\mathbf{x}_{K'}, r^1, \ldots, r^l) \right).$$

The last step is to marginalize over $r^1, \ldots, r^l$. This removes the factors $\sigma''(\mathbf{x}_{K'}, r^1, \ldots, r^l)$ and $\rho''(\mathbf{x}_{K'}, r^1, \ldots, r^l)$ and replaces them with a new factor $\psi_{K'}(\mathbf{x}_{K'})$ in l steps. This returns the result of the sum product of the columns $d_{u_1,v_1}, \ldots, d_{u_l,v_l}$ of the KroneRelation $\mathbf{D}$ for every combination of the variables $\mathbf{x}_{K'}$:

$$\phi''''(\mathbf{x}_f) \qquad = \sum_{x_{f+1}} \cdots \sum_{x_n} \left(\prod_{S \in \mathcal{E}'''' \setminus \{K'\}} \psi_S(\mathbf{x}_S) \right) \cdot \psi_{K'}(\mathbf{x}_{K'}).$$

From this point on, the marginalization continues for all $x_{f+1}, \ldots, x_n$ exactly as described in [AKNR16] and [Olt23].

Chapter 8

Related Work

8.1 Time Series Databases

Relational database management systems are usually deemed unsuitable for storing and processing time series data [DF14]. Therefore, many specialized time series databases have been developed.

A major challenge of time series databases is reducing the number of data points to be stored. The simplest and most common approach is data sampling, where only every n-th datapoint is stored, or data downsampling, where multiple datapoints are aggregated into one datapoint by taking the mean or some other aggregation function. The most popular like InfluxDB [Zeh17] and RRDTool [Oet05] usually use these approaches, by removing more data points for older data.

Another approach is to use a linear approximation of the data by connecting selected data points with straight lines. These data points can be uniformly spaced or selected by some important measurement. [Fu11]

More advanced methods represent time series in the transform domain. These methods include the Discrete Fourier Transform (DFT), the Principle Component Analysis (PCA) and Singular Value Decomposition (SVD) [KJF97]. [Fu11]

While the Kronecker decomposition is also computed using the SVD, it has the advantage that it can exploit the repeating structure of periodic time series. This will be discussed in much greater detail in Section 2.1.

8.2 Applications of the Kronecker Decomposition

Decomposing a matrix into one or more Kronecker products is sometimes called Kronecker product decomposition [PBC18a, HTMN22, TWL+20] or Kronecker decomposition [TCN+22]. Finding the best possible decomposition is called the Nearest Kronecker Product (NKP) problem by Loan [Loa00].

The Kronecker decomposition is heavily used in the field of audio and signal processing [PBC18b, EIPBC19, ZHL+17].

Another field where the Kronecker decomposition has been used in recent years is neural network compression, which aims to reduce the size of neural networks by training small networks to mimic larger networks or ensemble models [BCNM06, HVD15]. The main idea behind model compression using the Kronecker decomposition is to decompose the weight matrices of the neural network into a linear combination of Kronecker products of smaller matrices. This has been done for fully connected layers [ZW15], recurrent neural networks (RNNs) [TBG+20], language models [TWL+20], convolutional neural networks (CNNs) [HTMN22], and transformer networks [TCN+22].

To the best of our knowledge, the Kronecker decomposition has not been used to compress dense data in database relations.

Chapter 9

Future Work and Open Ends

9.1 Tensor Representation and Neural Networks

In this thesis, we used the relational representation for data matrices in relations. This means that the data matrix D is represented in the relation $\mathbf{D}$ by the columns $d_1, \ldots, d_n$. Together with the *rid* column, this results in the relation $\mathbf{D}(rid, d_1, \ldots, d_n)$, where the value of the i-th row and j-th column is the value of the attribute d_j in the tuple with the *rid* value i.

Another way to represent the data matrix D is to use the tensor representation, where the data relation $\mathbf{D}$ only has a single data column d and an additional *cid* column which represents the column index of the data. This results in the relation $\mathbf{D}(rid, cid, d)$, where the value of the i-th row and j-th column is the value of the attribute d in the tuple with the *rid* value i and the *cid* value j.

One could also use the tensor representation to represent KroneRelations and this would have certain advantages and disadvantages compared to the relational representation. A disadvantage is that some queries become more complex because the column needs to be fixed in the query to do column-wise aggregates. It is also less natural to store the data in this way in relational databases. An advantage of the tensor representation is that the indexing of rows and columns is done using the same mechanism, which better reflects the Kronecker product as a concept in linear algebra. Another advantage is that using the tensor representation, it is trivial to do a matrix multiplication in SQL using the query:

```
SELECT A.rid, B.cid, SUM(A.v * B.v) AS v
FROM A, B
WHERE A.cid = B.rid;
```

where v is the value column of the relations A and B.

During the work on this thesis, we also investigated the use of KroneDB to store and query neural networks. For this purpose, it was necessary to implement the matrix multiplication in SQL, which was based on the tensor representation. A fully connected layer can be represented as a matrix-vector product $D\mathbf{x}$ where D is the weight matrix and $\mathbf{x}$ is the input vector. In [TCN+22] it is shown that this can be done efficiently using the Kronecker decomposition, because

$$D\mathbf{x} = (S \otimes P)\mathbf{x} = vec(P\ vec^{-1}_{n_p,n_s}(\mathbf{x})S^\top),$$

where *vec* is the vectorization operator discussed in Section 2.1. We used this approach to implement a fully connected neural network as a series of SQL queries in KroneDB, but did not investigate further after our initial experiments did not show any promising results.

It should be further investigated if the advantages of the tensor representation outweigh the disadvantages and if it is worth it to use the tensor representation in KroneDB. Especially the question of the tensor representation together with the Kronecker decomposition being used to efficiently implement simple neural networks in SQL should be investigated further.

9.2 Implementation

The implementation of KroneDB is still in an early stage and was mainly developed to evaluate the concepts presented in this thesis. Therefore, it would be beneficial to improve the implementation in the future, such that it can be used to automatically decompose and query relations in a database with many other relations, which can be decomposed or not.

9.3 Arbitrary Datasets

The Kronecker decomposition is a general method to decompose matrices into Kronecker products. It is not limited to the decomposition of time series data. Even if there are no obvious periodic repeating patterns in the data, the Kronecker decomposition might still exploit structure in the data to reduce the number of parameters needed for a good approximation. We already discussed in Section 4.5 that the Kronecker decomposition can exploit structure in both dimensions of the data matrix, not just the temporal dimension.

This suggests that there are many more suitable datasets for the Kronecker decomposition, outside of time series data. In our successful experiments, the data has a natural temporal order. We also did experiments with machine learning datasets, over unordered relational data, where the Kronecker decomposition did not perform well.

It would be interesting to investigate further which datasets are suitable for the Kronecker decomposition and which are not.

9.4 Comparison to Specialized Database Systems

This thesis shows that the Kronecker decomposition improves the runtime of queries on time series in a relational database management system (RDBMS). It does not compare the performance of KroneDB to specialized time series databases like InfluxDB [Zeh17] or RRDTool [Oet05]. It might be interesting to see how KroneDB compares to these specialized databases in terms of performance and storage space, to evaluate the approach of leveraging RDBMSs to store and query time series data.

9.5 Special Index Columns

There can be some examples where it makes sense to avoid the abstraction of the *rid* column and use the real index column instead. This works if the original index column is easily divided into a scaling index and a period index. We can show this with our motivating example, the temperature data from Kloten ZH (Table 3.1), where the index column is the `time` column.

If we take a period length of 1 day, we can very nicely divide the `time` column into a scaling index and a period index. The scaling index would be the date and the period index would be the time of day. This means, that selecting a time involves splitting the requested time into the date and the time of day and directly using these two values to select or join on the scaling- and period-matrix.

Thus, to select all rows where the time is "01.01.2018 12:00", we can use the date "01.01.2018" to index the scaling-matrix and the time of day "12:00" to index the period-matrix. The FAQ for the original selection query would look like this:

$$\phi(time, d) = \psi_{time,d}(time, d) \cdot (time = \text{"01.01.2018 12:00"}).$$

This query can be rewritten to work with the decomposed data by splitting the time into the date and the time of day:

$$\kappa(date, \Delta time, d) = \\ \sigma(date, s) \cdot \rho(time, p) \cdot (d = s \cdot p) \cdot (date = \text{"01.01.2018"}) \cdot (\Delta time = \text{"12:00"}).$$

This also allows us to do some more interesting queries very efficiently, like selecting all rows where the time is 07:00:

$$\kappa(date, \Delta time, d) = \sigma(date, s) \cdot \rho(\Delta time, p) \cdot (d = s \cdot p) \cdot (\Delta time = \text{"07:00"}).$$

If we want to select a range of values that starts and/or ends in the middle of a period, we can use the same approach, but we need to split the requested range into multiple ranges for full periods and partial periods. To select all rows where the time is in the range [01.01.2018 12:00, 01.02.2018 12:00),

$$\phi(time, d) = \psi_{time,d}(time, d) \cdot (\text{"01.01.2018 12:00"} \leq time < \text{"01.02.2018 12:00"}),$$

we need to split the time range into three parts. The first part would be the range [01.01.2018 12:00, 02.01.2018 00:00), the second part would be the range [02.01.2018 00:00, 01.02.2018 00:00) and the third part would be the range [01.02.2018 00:00, 01.02.2018 12:00):

$$\begin{aligned}
\kappa_1(date, \Delta time, d) &= \sigma(date, s) \cdot \rho(\Delta time, p) \cdot (d = s \cdot p) \cdot \\
&\quad (date = \text{"01.01.2018"}) \cdot (\text{"12:00"} \leq \Delta time < \text{"24:00"}) \\
\kappa_2(date, \Delta time, d) &= \sigma(date, s) \cdot \rho(\Delta time, p) \cdot (d = s \cdot p) \cdot \\
&\quad (\text{"02.01.2018"} \leq date < \text{"01.02.2018"}) \\
\kappa_3(date, \Delta time, d) &= \sigma(date, s) \cdot \rho(\Delta time, p) \cdot (d = s \cdot p) \cdot \\
&\quad (date = \text{"01.02.2018"}) \cdot (\text{"00:00"} \leq \Delta time < \text{"12:00"}).
\end{aligned}$$

This can of course be generalized to any period length and the index column does not have to be a time column. As long as the values are sorted by the index column, and the delta between two consecutive values is constant, we can use the same approach. All we need to know is the first index value $index_0$, the delta between two consecutive index values $\Delta index$, and the period length m_p. This means that the index of the scaling-matrix is the first value of each period and the index of the period-matrix is the delta from the start of the period to the requested index value:

$$\begin{aligned}
\mathbf{index}_{scaling} &= [index_0 + (i-1)\Delta index \cdot m_p | i \in 1, \ldots, m_s] \\
\mathbf{index}_{period} &= [(i-1)\Delta index | i \in 1, \ldots, m_p]
\end{aligned}$$

, where $m_s = \frac{m}{m_p}$ is the length of the scaling-matrix and m is the length of the timeseries.

This indexing scheme could be very useful in specific applications and it would be interesting to investigate this further.

Chapter 10

Conclusion

The thesis concludes that the Kronecker decomposition can be used to efficiently store and query time series data with a periodic structure.

It allows to save storage space by providing a compression method that exploits the structure in the data and improves the approximation accuracy compared to other compression methods. Many data-dependent choices influence the quality of the approximation and need to be chosen carefully. The main choice that is heavily dependent on the data is the period length of the Kronecker decomposition. The best case is if the data has a natural period length, like the day cycle in weather or traffic data. Then, the period length can be chosen to be the length of the natural period or a multiple of it. Other choices include if certain columns or keys should be decomposed together and if the data should be normalized or shifted before the decomposition.

Querying the decomposed data directly is possible and efficient, especially for aggregation queries. It is explained how SQL queries can be rewritten to use the decomposed relations instead of the original data in any relational database management system (RDBMS) and immediately benefit from the more efficient querying.

Updates to the data can be handled by updating the decomposed relations directly and very efficiently, which also allows for streaming data to be decomposed and queried. During these updates, the new data points can automatically be completed using the learned structure of the data and anomalies can be detected and handled automatically.

It is shown how the KroneDecomposition can be used inside the FAQ framework. The hyperedges for KroneRelations can simply be substituted with new hyperedges for the decomposed relations and the InsideOut algorithm can be used to query the decomposed relations efficiently. This integration into the FAQ framework allows for the use of our work across a wide range of fundamental problems.

A system called KroneDB is implemented to demonstrate the concepts presented in this thesis. Experiments using this system on real-world data show that the theoretical advantages of the Kronecker decomposition can be realized in practice. The results are very promising and suggest that traditional RDBMSs might be able to rival specialized databases in the future.

Bibliography

[AKNR16] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. Faq: Questions asked frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '16, pages 13–28, New York, NY, USA, Jun 2016. Association for Computing Machinery.

[AMF06] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, page 671–682, New York, NY, USA, June 2006. Association for Computing Machinery.

[AW10] Hervé Abdi and Lynne J. Williams. Principal component analysis. *WIREs Computational Statistics*, 2(4):433–459, July 2010.

[BCNM06] Cristian Buciluǎ, Rich Caruana, and Alexandru Niculescu-Mizil. Model compression. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '06, page 535–541, New York, NY, USA, August 2006. Association for Computing Machinery.

[DF14] Ted Dunning and Ellen Friedman. *Time series databases: new ways to store and access data*. O'Reilly Media, Sebastopol, CA, 2014.

[EIPBC19] Camelia Elisei-Iliescu, Constantin Paleologu, Jacob Benesty, and Silviu Ciochina. A recursive least-squares algorithm based on the nearest kronecker product decomposition. In *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4843–4847, May 2019.

[Enc] Apache parquet. https://parquet.apache.org/docs/file-format/data-pages/encodings/ [accessed: 16.10.2023].

[Fag02] Ronald Fagin. Combining fuzzy information: an overview. *ACM SIGMOD Record*, 31(2):109–118, June 2002.

[Fu11] Tak-chung Fu. A review on time series data mining. *Engineering Applications of Artificial Intelligence*, 24(1):164–181, February 2011.

[HTMN22] Marawan Gamal Abdel Hameed, Marzieh S. Tahaei, Ali Mosleh, and Vahid Partovi Nia. Convolutional Neural Network Compression through Generalized Kronecker Product Decomposition. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(1):771–779, June 2022.

[HVD15] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. (arXiv:1503.02531), March 2015. arXiv:1503.02531 [cs, stat].

[KJF97] Flip Korn, H. V. Jagadish, and Christos Faloutsos. Efficiently supporting ad hoc queries in large datasets of time sequences. *ACM SIGMOD Record*, 26(2):289–300, June 1997.

[KL80] V. Klema and A. Laub. The singular value decomposition: Its computation and some applications. *IEEE Transactions on Automatic Control*, 25(2):164–176, April 1980.

[LMF+16] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. Data blocks: Hybrid oltp and olap on compressed storage using both vectorization and compilation. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 311–326, New York, NY, USA, June 2016. Association for Computing Machinery.

[Loa00] Charles F. Van Loan. The ubiquitous Kronecker product. *Journal of Computational and Applied Mathematics*, 123(1):85–100, November 2000.

[Met] Federal office of meteorology and climatology meteoswiss. https://www.meteoschweiz.admin.ch.

[Oet05] Tobias Oetiker. Rrdtool - the time series database. http://www.rrdtool.org/, 2005.

[Olt20] Dan Olteanu. The relational data borg is learning. (arXiv:2008.07864), August 2020. arXiv:2008.07864 [cs].

[Olt23] Dan Olteanu. Efficient Algorithms - Sloving Functional Aggregate Queries. University of Zürich, http://www.ifi.uzh.ch/en/dast/teaching/EA.html [accessed: 13.10.2023], 2023.

[OS16] Dan Olteanu and Maximilian Schleich. Factorized databases. *ACM SIGMOD Record*, 45(2):5–16, Sep 2016.

[PBC18a] Constantin Paleologu, Jacob Benesty, and Silviu Ciochină. Linear System Identification Based on a Kronecker Product Decomposition. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 26(10):1793–1808, October 2018.

[PBC18b] Constantin Paleologu, Jacob Benesty, and Silviu Ciochină. Linear system identification based on a kronecker product decomposition. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 26(10):1793–1808, October 2018.

[RM19] Mark Raasveldt and Hannes Mühleisen. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1981–1984, New York, NY, USA, June 2019. Association for Computing Machinery.

[RV22] Tim Roughgarden and Gregory Valiant. CS168: The Modern Algorithmic Toolbox Lecture #9: The Singular Value Decomposition (SVD) and Low-Rank Matrix Approximations, April 2022.

[RVH93] Mark A. Roth and Scott J. Van Horn. Database compression. *ACM SIGMOD Record*, 22(3):31–39, September 1993.

[TBG+20] Urmish Thakker, Jesse Beu, Dibakar Gope, Chu Zhou, Igor Fedorov, Ganesh Dasika, and Matthew Mattina. Compressing rnns for iot devices by 15-38x using kronecker products. (arXiv:1906.02876), January 2020. arXiv:1906.02876 [cs, stat].

[TCN+22] Marzieh Tahaei, Ella Charlaix, Vahid Nia, Ali Ghodsi, and Mehdi Rezagholizadeh. KroneckerBERT: Significant Compression of Pre-trained Language Models Through Kronecker Decomposition and Knowledge Distillation. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2116–2127, Seattle, United States, July 2022. Association for Computational Linguistics.

[TWL+20] Urmish Thakker, Paul N. Whatmough, Zhi-Gang Liu, Matthew Mattina, and Jesse Beu. Compressing Language Models using Doped Kronecker Products, November 2020.

[UTD] UTD19 - largest multi-city traffic dataset publically available. https://doi.org/10.1038/s41598-019-51539-5/.

[Vel14] Todd Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm, 2014.

[Yan81] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Very Large Data Bases, International Conference on Very Large Data Bases*, 01 1981.

[Zeh17] Syeda Noor Zehra. Time series databases and influxdb. 2017.

[ZHL+17] Guangxu Zhu, Kaibin Huang, Vincent K. N. Lau, Bin Xia, Xiaofan Li, and Sha Zhang. Hybrid beamforming via the kronecker decomposition for the millimeter-wave massive mimo systems. *IEEE Journal on Selected Areas in Communications*, 35(9):2097–2114, September 2017.

[ZW15] Shuchang Zhou and Jia-Nan Wu. Compression of fully-connected layer in neural network by kronecker product. (arXiv:1507.05775), July 2015. arXiv:1507.05775 [cs].

Usage of generative AI

In agreement with Prof. Dr. Dan Olteanu, the following forms of generative AI were used in this thesis:

1. Github Copilot and ChatGPT were used as coding assistance in the implementation of the software.
2. Spellchecking, grammar checking, and style checking were done using Grammarly, Github Copilot, and ChatGPT.
3. Writing LaTeX code to create figures, tikz diagrams, and matrices was assisted by Github Copilot and ChatGPT.

Chapter 11

Appendix

In this chapter, the SQL query for the sum product is shown in Figure 11.1. Some additional runtime results are shown in Figures 11.2, 11.3, 11.4, 11.5, 11.6, 11.7 and 11.8.

```
SELECT x, D_1.d_1 + D_2.d_2 + D_3.d_3 + D_4.d_4 AS d
FROM (
    SELECT x, S_1.s_1_prime * P_1.p_1_prime AS d_1
    FROM (
        SELECT x, SUM(s_1 * s_2) AS s_1_prime
        FROM (
            SELECT x, rid_s, s_1
            FROM scaling
            WHERE r = 1
        ) JOIN (
            SELECT x, rid_s, s_2
            FROM scaling
            WHERE r = 1
        )
        GROUP BY x
    ) AS S_1 JOIN (
        SELECT x, SUM(p_1 * p_2) AS p_1_prime
        FROM (
            SELECT x, rid_p, p_1
            FROM period
            WHERE r = 1
        ) JOIN (
            SELECT x, rid_p, p_2
            FROM period
            WHERE r = 1
        )
        GROUP BY x
    ) AS P_1
) AS D_1 JOIN (
    SELECT x, S_2.s_2_prime * P_2.p_2_prime AS d_2
    FROM (
        SELECT x, SUM(s_1 * s_2) AS s_2_prime
        FROM (
            SELECT x, rid_s, s_1
            FROM scaling
            WHERE r = 1
        ) JOIN (
            SELECT x, rid_s, s_2
            FROM scaling
            WHERE r = 2
        )
        GROUP BY x
    ) AS S_2 JOIN (...) AS P_2
) AS D_2 JOIN (...) AS D_3 JOIN (...) AS D_4;
```

Figure 11.1: SQL query for the sum product
`SELECT x, SUM(d_11 * d_22) AS d FROM data GROUP BY x;`
over a rank-2 KroneDecomposition.

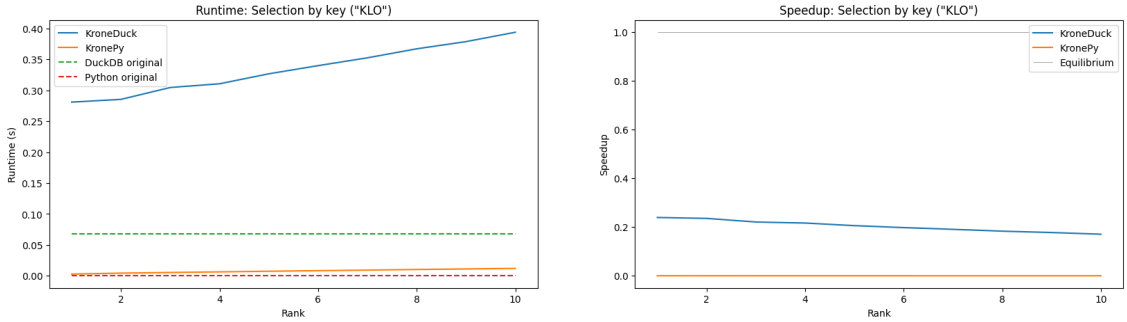


Figure 11.2: Runtime and speedup of a selection query on the key column. The selected key is `KLO`.

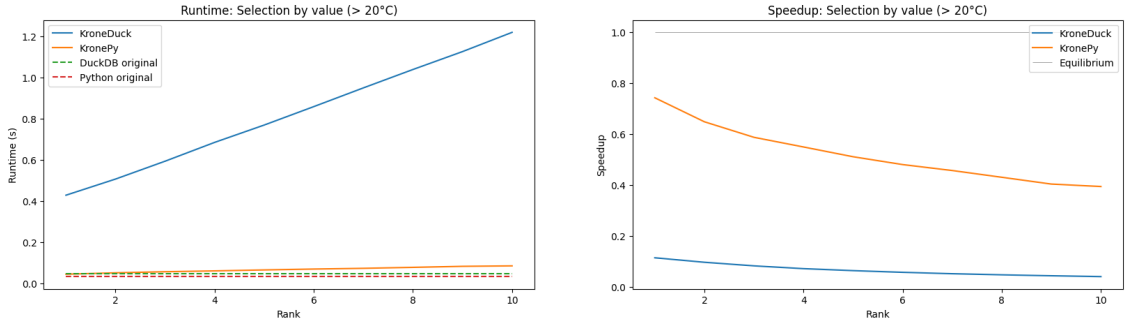


Figure 11.3: Runtime and speedup of a selection query on the value column. The selected value is > 20°C on the only value column `Temperature_2m (°C)`.

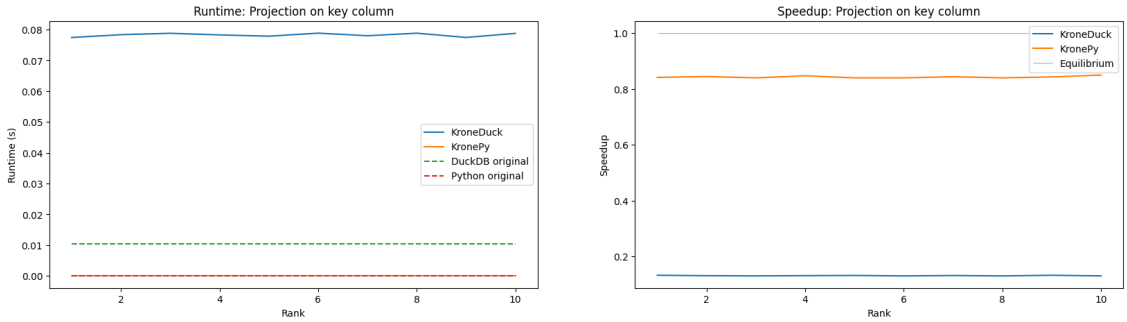


Figure 11.4: Runtime and speedup of a projection query onto the key column.

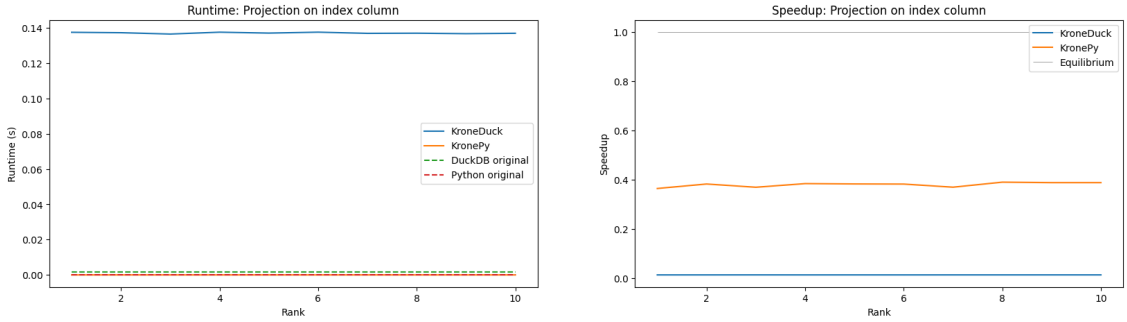


Figure 11.5: Runtime and speedup of a projection query onto the index column.

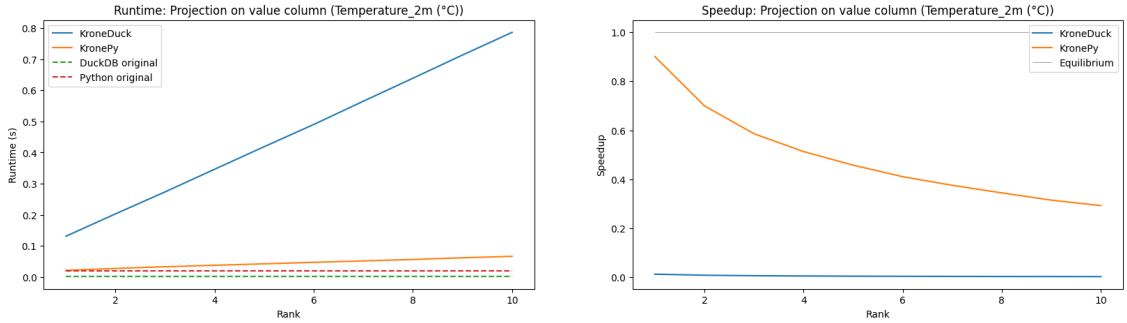


Figure 11.6: Runtime and speedup of a projection query onto the value column.

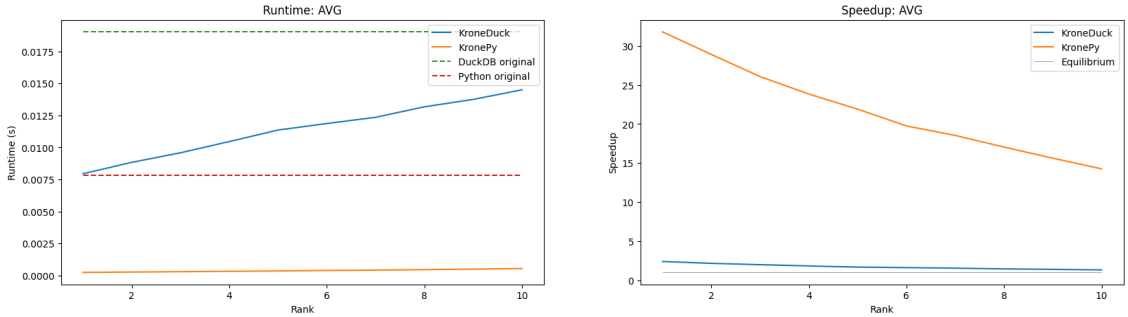


Figure 11.7: Runtime and speedup of an average query on the value column.

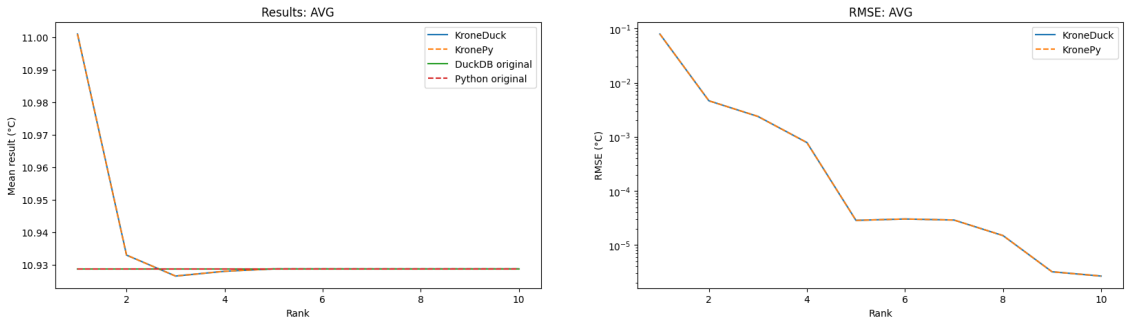


Figure 11.8: Mean result and root mean squared error (RMSE) of an average query on the value column.