University of Zurich[UZH]

Department of Informatics

BSc Thesis

Scalable Exploratory Analyses of Feed Data

Tobias Egger

Matriculation Number: 16-728-016

Email: tobias.egger@uzh.ch

February 4, 2021

supervised by Prof. Dr. Michael Böhlen

Acknowledgements

At this point I would like to express my sincere gratitude to my supervisor Prof. Dr. Michael Böhlen. In our numerous, mainly digital meetings he provided me with valuable feedback. The support of Prof. Dr. Michael Böhlen helped a lot in the process of writing the thesis. Additionally, I am very grateful for the opportunity to write the thesis at the Database Technology Group of the University of Zurich.

Abstract

In this thesis, an approach to measuring performance in a multi-tier application is designed and implemented. The results of the performance measurements are used to identify the main limiting factors in the Swiss Feed Database. The problematic parts in the PostgreSQL database tier are subsequently optimized. For this, query plans are analysed in detail and different performance-enhancing features in PostgreSQL are used. Spatial indexing is used to optimize spatial joins. Just-in-time compilation enhances the overall performance of database queries. Furthermore, temporary tables are used as a substitute for common-table expressions. Through a reduction of output size, the performance is enhanced as well. Finally, the substitution of slow string functions helps to leverage the performance. The optimizations are evaluated regarding the performance-gain and the scalability.

Zusammenfassung

In dieser Arbeit wird ein Ansatz zur Performanzmessung in einer Multi-Tier-Applikation entworfen und implementiert. Die Ergebnisse der Performanzmessungen werden verwendet, um die wichtigsten limitierenden Faktoren in der Schweizer Futtermitteldatenbank zu identifizieren. Die problematischen Teile in der PostgreSQL-Datenbank werden anschliessend optimiert. Dazu werden die Auswertungspläne im Detail analysiert und verschiedene geschwindigkeitsverbessernde Eigenschaften in PostgreSQL genutzt. Spatial Indexing wird verwendet, um räumliche Joins zu optimieren. Die Just-in-Time-Kompilierung steigert die Gesamtperformanz von Datenbankabfragen. Im Weiteren werden temporäre Tabellen als Ersatz für Common-Table-Expressions verwendet. Durch eine Reduzierung der Resultatgrösse wird die Performanz ebenfalls verbessert. Schliesslich hilft die Ersetzung von langsamen String-Funktionen, die Performance zu steigern. Die Optimierungen werden hinsichtlich des Performanzgewinns und der Skalierbarkeit bewertet.

Contents

1. Introduction **10**
- 1.1. Performance Analysis . . . 10
- 1.2. Performance Optimization . . . 10
- 1.3. Evaluation of the Improvements . . . 11

2. Background **12**
- 2.1. Architecture . . . 12
- 2.2. Performance Analysis in a Multi-Tier Application . . . 12
  - 2.2.1. Theory . . . 12
  - 2.2.2. Designing the implementation . . . 14
- 2.3. Performance Analysis in the Swiss Feed Database . . . 17
  - 2.3.1. Test Preparation and Execution . . . 20
  - 2.3.2. Log Formats . . . 20
  - 2.3.3. Log Preprocessing . . . 22
- 2.4. Database Optimization . . . 25
  - 2.4.1. Query Plans . . . 26
  - 2.4.2. Finding the Bottle-Neck in a Query Plan . . . 26
  - 2.4.3. PostgreSQL's Performance Enhancing Features . . . 27
  - 2.4.4. Spatial Indexes . . . 29
  - 2.4.5. Spatial Joins . . . 30
  - 2.4.6. Common Table Expression (CTE) vs. Temporary Table . . . 31

3. Bottle-Neck Assessment **33**
- 3.1. Timestamp Analysis . . . 33
- 3.2. Client UI Component Analysis . . . 34
- 3.3. Summary . . . 35
- 3.4. Validation of Analysis . . . 35
  - 3.4.1. Validation-Run 1: Blocking Code . . . 35
  - 3.4.2. Validation-Run 2: Cache database results . . . 36

4. Chart UI Component Optimization **38**
  - 4.0.1. Evaluation . . . 41

5. Table UI Component Optimization **42**
- 5.1. Optimization for Detail Queries . . . 43
  - 5.1.1. Evaluation . . . 46

5.2. Optimization for Summary Queries 46
5.2.1. Part 5 46
5.2.2. Part 2 50

6. Evaluation of the optimization **52**
6.1. Overall Performance 52
6.2. Scalability 53

7. Conclusion **55**
7.1. Future Work 56

Bibliography **57**

Appendices **60**

A. Glossary **60**
A.1. Query Types 60

B. Performance Analysis **62**
B.1. System Component Performance Test-run 62
B.2. Statistical Analysis 64

C. SQL Statements **65**

List of Figures

2.1. Basic Architecture of the feedbase web application 12
2.2. Sequence diagram of a multi-tier application 13
2.3. Sequence Diagram of a query in the feedbase 19
2.4. Illustration to show the construction of a spatial index, source: [8] 29
2.5. Illustration to show a spatial join between rectangles and lines 30

3.1. Mean of the differential duration for all test-runs 33
3.2. Test-run 2: Duration of timestamp D for UI components for different queries 34
3.3. Test-Run 2: Duration of timestamp A for UI components for different queries 35
3.4. Validation-Run 1: Block Map in D 37
3.5. Validation-Run 2: Performance Measurement with Database Caching 37

4.1. Execution Time of Chart Query Parts displayed with a logarithmic scale. . . . 38
4.2. Measurement of the duration of chart query parts before and after optimization displayed with a logarithmic scale. 41

5.1. Duration of database queries for the table UI component. 42
5.2. Duration of database queries for the table UI component on PostgreSQL12. . 44
5.3. Measurement of the duration of the table UI component for detail queries before and after the optimization. 46
5.4. Execution Time of Summary Query Parts 47
5.5. Measurement of the duration of the table UI component for summary queries before and after the optimization of part 5. 49
5.6. Measurement of the duration of the table UI component for summary queries before and after the optimization of part 2. 51

6.1. Difference between durations of system components before and after the optimization. 52
6.2. Mean of the duration of all timestamps before and after the optimization. . . . 53
6.3. Results of a scalability test for database queries before and after optimization. 54

B.1. Test-run 1: Performance of system components during different queries . . . 62
B.2. Test-run 2: Performance of system components during different queries . . . 62
B.3. Test-run 3: Performance of system components during different queries . . . 63

List of Tables

2.1. Description of all fields in a log entry from the application log. 20

3.1. Amount of times a timestamp had the longest duration in a query 33
3.2. Queries selected for the validation-runs 36

5.1. Example for the problematic join in the SQL statement for SummaryResults . 48
5.2. Result of the optimized join in the SQL statement for SummaryResults 49

A.1. Summary Queries and their ID's 60
A.2. Detail Queries and their ID's 61

B.1. Statistical parameters for the differential duration of all three test-runs 64

Listings

2.1. Example for the content of an application log file. 21
2.2. Source: PostgresQL documentation [1] 21
2.3. Example of an ordered application log file 23
2.4. Example of an unordered application log file 23
2.5. Example of a correctly formatted log array used as input for restructuring . . 24
2.6. Restructured log entries from listing 2.4 24
2.7. Example of a query plan with a bottle-neck created with EXPLAIN 27
2.8. Excerpt of a query plan using parallel features, created with EXPLAIN 29

4.1. Query Plan of the join leading to slow performance 39
4.2. Subquery of part 7 responsible for the long duration 39
4.3. Query Plan for the optimized Join 41

5.1. Excerpt from the Query Plan for DetailResults in PostgreSQL 13 43
5.2. Query Plan Summary for DetailResults in PostgreSQL 13 43
5.3. Wrong Estimates in the Query Plan for DetailResults in PostgreSQL 12 . . . 44
5.4. Excerpt of a Query Plan for DetailResults in PostgreSQL 12 using temporary tables instead of CTE's 45
5.5. SQL statement for SummaryResults 47
5.6. Excerpt of a Query Plan for SummaryResults 50
5.7. Subquery of part 2 where a scan with a long duration is performed 50
5.8. Optimized creation of a list of ids from a string of ids 51

C.1. Chart query with a duration of over 20s 65
C.2. DetailResults query with a duration of around 6s 67
C.3. SummaryResults query 68

1. Introduction

The Swiss Feed Database allows users to query feed data in an online application. While in the past a lot of work was done in extending and improving functionalities, the overall performance of the Swiss Feed Database is not satisfying. When querying specific feed data, users have to wait up to 30 seconds until the results are displayed. This thesis improves the performance of the Swiss Feed Database so that the user experiences fewer interruptions and the interaction gets more fluent.

1.1. Performance Analysis

The first topic of the bachelor thesis was the analysis of exploratory queries in the Swiss Feed Database. The focus was on finding the main limiting parts in the system regarding the performance of the first visualization of a query result. The assessment and evaluation were based on a comprehensive and architecture-dependent understanding of the system. For this, a concept applicable to multi-tier applications in general was developed. Dominique Hässig already improved the performance of the Swiss Feed Database in the frontend. In her thesis, she points out that further optimization needs to be done in the PostgreSQL database [2]. The hypothesis for the performance evaluation was: The main limiting part regarding performance is the PostgreSQL database.

The measurements done in this thesis support the hypothesis. The average duration of different system components was used to get a rough overview of how the response time is distributed between the system components. At around 1.4 seconds, the average duration of the PostgreSQL database was much higher than the average duration of the other system components, which had average durations ranging from 0.05 seconds to 0.5 seconds. Additionally, in most of the queries, the PostgreSQL database was the component with the longest duration of all components. In a second step, the distribution of the database durations over the different user interface components (map, chart and table) was evaluated. It was found, that the chart and the table contribute the most to the long durations of the database. The map user interface component hardly ever showed durations of over one second.

1.2. Performance Optimization

After the evaluation of the performance and the identification of the main limiting factors, the second topic was to design and implement optimizations to enhance the performance of the Swiss Feed Database. The target was that every component, used for the first visualization of a query result in the Swiss Feed Database, had a duration of around one second.

Based on the results in the analysis the focus of the optimization was on database queries used for the chart and the table visualization. For the optimization the execution plans of the database were analysed in detail, searching for the limiting parts.

During the optimization, it was discovered that indexes are crucial for performant spatial joins. In the database query for the chart visualization, a join of two tables, involving some spatial features, was identified as the blocking part. Creating an index on the spatial features helped the database to use a more efficient method of joining. As a result, the duration of the database query dropped from around 24 seconds to around a second.

Furthermore, it was shown that keeping the database system up-to-date can have a big impact on the performance of the system. It is advisable to always keep the database system up-to-date. In the Swiss Feed Database, the table visualization was accelerated due to new performance-enhancing features in the updated version of the database system.

In complex queries, the query is often split into parts. This can be done by using Common Table Expressions or temporary tables. It was observed that Common Table Expressions are not as scalable as temporary tables. The duration of certain queries was reduced from 4.2s to 2s, solely by replacing Common Table Expressions by temporary tables.

While a lot of optimization can be done using the query plans, sometimes it is needed to go one step further. The next step is to analyse the semantics of the query in detail. Understanding the query semantics can help to find inefficiencies on a higher, conceptual level. For example in one query it was found, that a join produced output, that was not used. The performance was enhanced by restricting the join condition.

Finally, it was shown that the impact of expression evaluation on the performance of queries can be very big. The replacement of slow string functions with more efficient functions enhanced the performance of the queries used for the table visualization a lot.

1.3. Evaluation of the Improvements

The last topic was the evaluation of the optimizations in the context of the Swiss Feed Database. Especially the performance-gain and the scalability of the solution were assessed.

The average duration of the PostgreSQL database dropped from around 1.4 seconds to around 0.4 seconds. Especially the very long durations were decreased a lot. The user experience is much more fluent and the user waits at most 2-3 seconds for the results. In all queries with long durations, bottle-necks could be identified and optimized. However, the target of all components having durations smaller than a second was not achieved.

The optimized database queries were tested with different table sizes to get a first impression of their scalability. The measurements showed no signs of a strong growing pattern, which is good to know. The tests indicate that even with six times more data than now, the optimized database queries will have durations < 4 seconds. This means that the database queries in the Swiss Feed Database can handle an increase in data well. However, it was not assessed how the other system components react to a growing data set.

2. Background

2.1. Architecture

The Swiss Feed Database, frequently called the feedbase, is a web application. The structure of such a system is shown in Figure 2.1. The design of the feedbase can be described as a three tier architecture. The client tier is the user interface through which the user interacts with the system and the system displays its data. The application tier holds the whole business logic and the algorithms to process user requests. The database tier saves and loads the data. The tiers communicate using the request-response principle. A tier issues a request to another tier and waits until the response is returned.

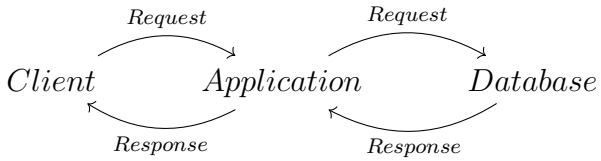


Figure 2.1.: Basic Architecture of the feedbase web application

The landing page of the feedbase offers the user some predefined queries. When a user clicks on a query the data for that specific query is visualized. The visualization can consist of different user interface components, further often called UI components. There is a map UI component showing a map of the area of interest, a chart UI component showing a scatter plot of the data and a table UI component showing further information about the data points.

The queries can be categorized into two types. First, there are detail queries, which are visualized with a table, a chart and a map UI component. Secondly, there are summary queries, which are visualized with a table UI component only. On the landing page, the summary queries are located in the left column and the detail queries are located in the right column. In tables A.1 and A.2 the predefined queries are categorized by query type.

2.2. Performance Analysis in a Multi-Tier Application

2.2.1. Theory

Performance is measured for different components of the system. Such a system component can be a single request, a part of a bigger computation or everything done in a tier. In figure 2.2 a sequence diagram of a schematic multi-tier application is shown. The application has two tiers: Tier A and Tier B. In the process shown in the sequence diagram Tier A first executes

a system component. Then Tier A makes a request to Tier B, which computes something and returns the response to Tier A.

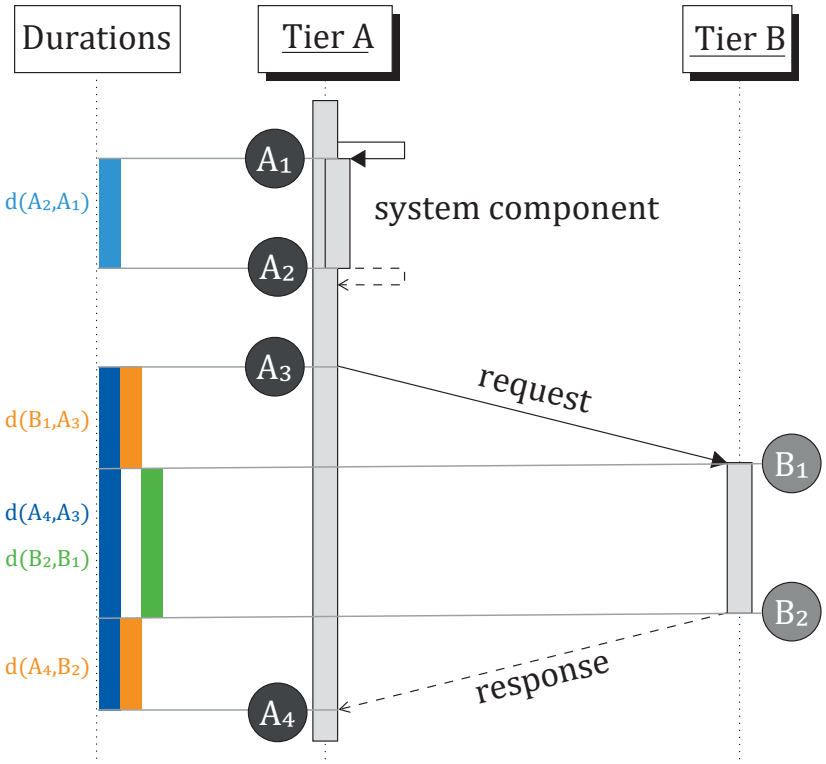


Figure 2.2.: Sequence diagram of a multi-tier application

The goal is to measure how much time different system components need to do their computations. Measuring time within a tier is pretty simple. An example is the system component in figure 2.2. As soon as the system component starts to work the current time (time A_1) is measured. When the system component has finished the computations the current time is measured again (time A_2). From these two measurements the duration the system component needed to respond to the request can be calculated: $d(A_2, A_1)$. In the sequence diagram, this duration is shown in light blue.

When the time that is spent between two tiers should be measured, e.g. how long a request takes to get from Tier A to Tier B, a new problem emerges. The problem is illustrated in figure 2.2. Using the approach from above the current time when the request was sent in one tier (time A_3) and the current time when the request is received in another tier (time B_1) would be measured. To calculate the duration based on two times measured on different tiers, it must be ensured that the two tiers have the exact same current time. This is a condition, that is very difficult to ensure, especially when the tiers run on different operating systems. In the sequence diagram above this would mean that calculating $d(B_1, A_3)$ and $d(A_4, B_2)$ is not possible unless it is known that the current time is exactly the same on Tier A and Tier B. The synchronisation of times could be done by using a dedicated time server to gather the current time in all tiers. The disadvantage of this approach is that it adds another tier to the system

and generates more overhead that could influence the measurements.

A simpler approach to this problem is to not measure the time of the request $d(B_1, A_3)$ or the response $d(A_4, B_2)$ on their own, but to measure the time of request and response together: $d(B_1, A_3) + d(A_4, B_2)$. This is a lot easier because now a different approach is possible. The time Tier A waited for Tier B is calculated as follows: $d(A_4, A_3)$. The time Tier B needed to respond to Tier A can be calculated as well: $d(B_2, B_1)$. Now the time Tier B needed is subtracted from the time Tier A waited. The result is the time that the request and response needed together: $d(A_4, A_3) - d(B_2, B_1)$. This approach has the disadvantage that the distribution of the calculated duration between request and response is not known. Nevertheless, this approach is less intrusive than the first approach. This led to the decision to choose the second approach for the feedbase.

The method described above assumes, that all tiers can measure the start and end time of the computations. Let's assume that Tier B in figure 2.2 is not able to measure the start and end time (e.g. because Tier B cannot be extended at all). In this scenario, it is no longer possible to measure the times B_1 and B_2. The only rough estimate one can measure for Tier B is $d(A_4, A_3)$. Because this duration includes the durations of the request and the response, the estimate gets worse the larger the durations of request and response ($d(B_1, A_3)$ and $d(A_4, B_2)$) are.

2.2.2. Designing the implementation

When implementing performance measuring in a multi-tier application it is important to think about the following aspects:

Which tiers are extendable to install code for the measurement? Are there restrictions?

Tiers that are implemented by oneself are often easily extendable. Tiers that are based on third party software, which is for example often the case for database systems, are not extendable easily. The simplest approach is to find a solution where only the configuration of the third party software has to be changed. This is straight forward, but the possibilities are restricted.

In the feedbase all tiers are extendable. This means that all tiers can be configured to measure the time of computations, either by inserting custom code or by changing the configuration. In the client and the application tier, custom code can be inserted nearly everywhere. In the database tier, there is almost no possibility to insert custom code. Therefore the extension is done by changing the configuration of the database system, e.g. to generate logs with information about the duration of statements.

How are the times measured?

Measuring times gets easier the more automated the process of measuring is. If you for example have to click through every UI component several times for one measurement, the method of measurement is time-consuming and has potential for automation. The goal is to have a

method where one needs to start the measurement and then all the actions are executed automatically. This will help to get more reliable measurements after all because the measurements are always done the same. Furthermore, with automation it is possible to execute each action multiple times, which helps to flatten natural fluctuations in the measurements. This can be done by calculating some statistical properties for each execution of an action, e.g. calculating the mean of all measurements for a particular action. When measuring the performance of actions in a system that are triggered through the user interface, the automation can be achieved by a user-simulator. Such a simulator can be as simple as a script, that goes through specific items in a user interface once or multiple times and clicks on them to trigger the desired action.

To efficiently assess the duration of multiple queries the feedbase is extended with a click-simulator, which goes through a specified set of queries on the startpage and executes each query three times. A test-run consists of preparation, execution, preprocessing and visualization. The execution is defined as one complete run of the click-simulator through all queries listed on the startpage of the feedbase. These queries get executed multiple times. Each execution is called an iteration. The procedure of test preparation and execution is described in section 2.3.1

Where and how are the measured times gathered?

Log files are a good concept to efficiently gather the measured times. In a first step, it should be determined how many log files are generated. Depending on the multi-tier architecture it can be reasonable to have a log file on each tier. With this solution writing to a log file is very easy, because the log file is located in the same tier. However for the evaluation of the measured times, the gathering of the measured times from each tier is much more complex. The format of the logs must include some log fields that allows combining the log entries from the different tiers. Another solution would be to have one log file for all tiers, which makes writing to the log file more complex. Nevertheless, all the measured times are already gathered in one place for the analysis. This simplifies the processing of the logs a lot because the step of combining all the log files is not necessary with this solution. Log files can adhere to a certain standard or a new log format can be created for the specific use case.

In the database tier of the feedbase there is no possibility to configure the database system to write logs to another tier [1]. Therefore the database tier has its own log. The database log adheres to the default convention from PostgreSQL with minor adjustments. Because sending messages from the client tier to the application tier is fairly easy in the feedbase, the choice was made to combine logs from those tiers in one log, located in the application tier. For that, the measurements from the client tier are sent to a dedicated endpoint in the application tier using AJAX. A new log format was defined for the application log, tailored to the needs of the measurements. The log formats are described further in section 2.3.2. How the log entries are combined and ordered is described in detail in section 2.3.3.

The combination of the database and the application logs in the feedbase is only possible because of a common identifier both logs share. This identifier is unique for each request to the database. It is needed to insert each database log entry into the correct position in the application log. The identifier is constructed out of four parts: the query id, the current iteration, the name of the UI component and the id of the database request (only needed for table queries,

because there are multiple database requests for the table UI component). An example could look like this: table/paginate-2 528-0. The identifier is placed at the beginning of each SQL statement the application tier issues. The database was configured so that it logs the duration and the actual SQL code of every executed SQL statement.

How are the times processed and visualized?

When evaluating measured times it is handy to have well processed and visualized data. If for example for each tier the measured times are gathered in a separate log file, the log files first need to be combined to have all the information in one place. Additionally, some computed metrics are often crucial to get information out of log files. In the end, looking at a log file is not very pleasing and correlations can easily be overseen, because of the sheer amount of unformatted, clumped data.

In the feedbase the logs are accessible through different endpoints in the API of the application tier. Each endpoint aggregates the logs differently, so that the requesting instance only has to do the visualization. The preprocessing is done every time one of those endpoints is called. It involves filtering the database logs, ordering the application and the database logs, inserting the database logs into the application logs and some final computation and aggregation. The details of the preprocessing are found in section 2.3.3.

REST-ful APIs

Tiers can have Application Programming Interfaces (APIs) designed in a REST-ful manner. One idea of a REST-ful API is that the API is not aware of the state the requesting tier is in. This can lead to problems when log entries from different tiers should be combined. If a client tier gets its data from the application tier over a REST-ful API, it must be ensured that the client tier adds information about its state to the request, so that the application tier can add the current state to its log entries. This additional information breaks the concept of a REST-ful API, but is necessary so that later the log of the application tier can be combined with the log of the client tier.

A simplified example of a request in the feedbase illustrates the problem. The sequence of actions leading to the problem is as follows:

1. The user clicks on the query with id 898.
2. The client tier fetches the necessary data from different endpoints. Each request has the same information for specifying the wanted data: *feed 505 with nutrients 42, 11 and 35 should be displayed*. The endpoints are:
   - /api/charts/hull: data for the scatterplot
   - /api/map/cantons: data for the map
   - /api/table/paginate: data for the table
3. The application tier processes the requests and logs the duration of the requests. Since the application tier doesn't know the query id, it logs:

- queryId: None, origin: application , description: /api/charts/hull , duration: 2.4
- queryId: None, origin: application , description: /api/map/cantons , duration: 1.2
- queryId: None, origin: application , description: /api/table/paginate , duration: 2

4. The data is returned to the client tier. After successful visualization the client tier logs:
   - queryId: 898, origin: client , description: visualization , duration: 5.6

The problem arises when the client log should be combined with the application log. Because in the application log the query id is not present, it is impossible to correctly combine the two logs. To solve this problem, the query id needs to be added to the information in the client request, e.g. in step 2: *feed 505 with nutrients 42, 11 and 35 should be displayed for the query with id 898*. Thus the application tier is aware of the query id and can add it to the logs. As a result, the combination of the logs is as simple as matching the query ids.

In the feedbase the application tier offers a REST-ful API, which is used by the client tier. This yields the problem, that the application tier never knows in which iteration of which query the client tier currently is. To associate every timestamp measured in the application tier with a query id and iteration it is thus necessary to extend the client tier by adding the current query id and iteration to each relevant request. This is done by adding the GET-parameters `queryId` and `iteration` to the requests to the API. With that, it is possible to access the query id and the iteration in the application tier. This is done over the request object `req` like so: `req.query.queryId` and `req.query.iteration`.

Asynchronous processes

Another problem arises when a system has asynchronous parts, e.g. a tier that issues multiple requests asynchronously to another tier. Because the asynchronous processes run in parallel, it is not possible to predict the order of the logs produced by these processes. It is therefore needed, that the logs are ordered according to the correct logical order before processing them.

In the feedbase this problem occurred with multiple requests from the client tier to the application tier. An example and the solution to this specific problem can be found in section 2.3.3.

2.3. Performance Analysis in the Swiss Feed Database

The sequence diagram in figure 2.3 illustrates the interaction of a user with the feedbase. The diagram captures the whole process involved from clicking on the query until all the data is visualized in the user interface. Based on the sequence diagram the measured system components are defined. For each system component, a timestamp is measured. A timestamp includes a start and an end time. The duration of a timestamp is defined as the difference between the start and end time of the timestamp. Each tier and the time which is spent between the tiers is measured. This yields the following timestamps:

Label	Location	Start	End
C	Client	When a user selects a query	When all components finished loading and are displayed
CA	Client-Application	When the data for a component is requested	When the data is received and a change is triggered in Angular
A	Application	When a request is received by the server	When the request is returned to the client
AD	Application-Database	When a request is sent to the database server	When the response is received from the database server
D	Database	When a statement starts execution on the database server	When the statement finishes execution on the database server

In the sequence diagram, the timestamps are displayed with coloured circles at their start and end time.

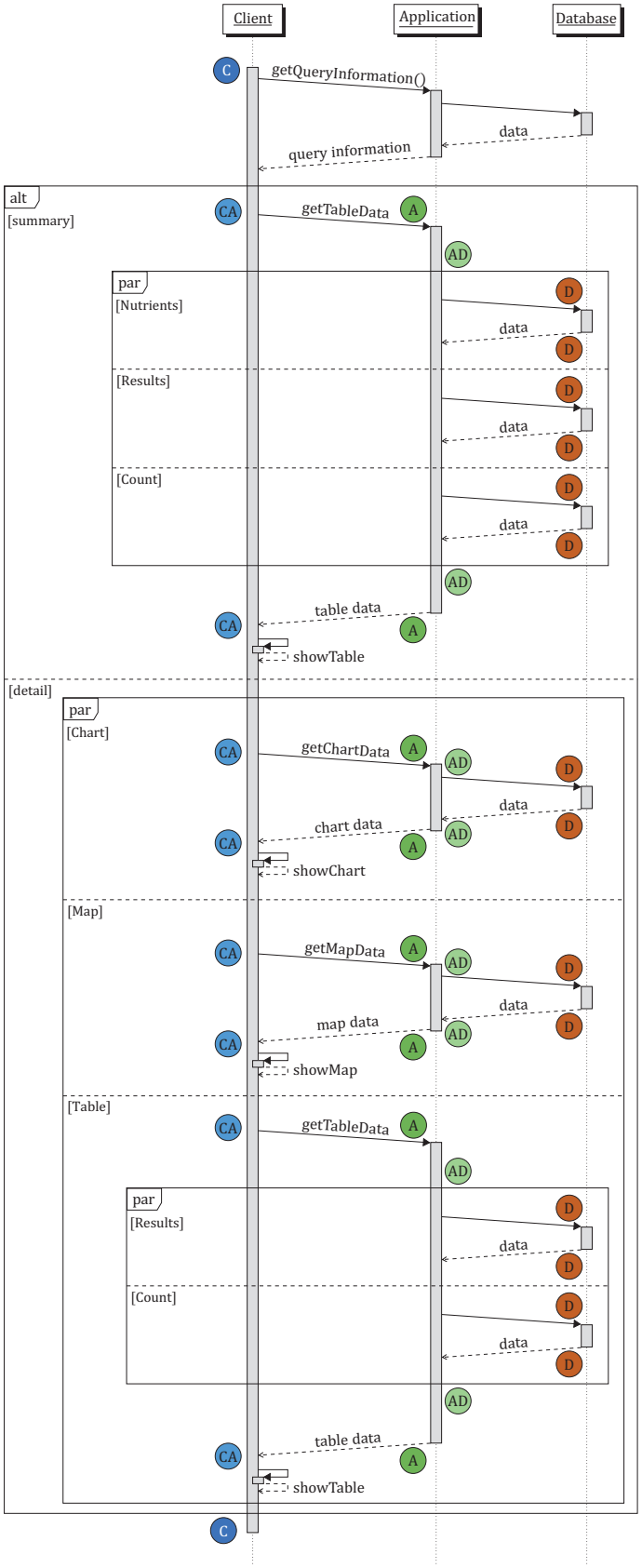


Figure 2.3.: Sequence Diagram of a query in the feedbase

2.3.1. Test Preparation and Execution

To prepare a test-run the current log entries in the application as well as on the database server are deleted. Deleting the application logs is achieved by calling the endpoint `DELETE /api/logs/files/stopwatch`. Deleting the database logs is done in the shell of the database server directly. After deletion, a restart of PostgreSQL is needed.

The click-simulator offers different options. It is possible to either run summary or detail queries. Another option is to define a set of queries to be executed.
To only run summary queries the following URL is used:
`http://130.60.24.196/?testType=summary&testOffset=0&testIteration=0`
For detail queries the URL changes to:
`http://130.60.24.196/?testType=detail&testOffset=0&testIteration=0`
To run a set of queries the following URL is used: (QID1,QID2 are the ids of the queries to be run, separated by commas)
`http://130.60.24.196/?testType=detail&testOffset=0&testIteration=0&testSet=QID1,QID2`
The URLs defined above can be entered in a browser and the click-simulator will run through the queries, calling each query three times. For a test-run, the URL for summary queries is called first. After the click-simulator finished execution of the summary queries, the URL for detail queries is called.

2.3.2. Log Formats

Application Log

The application log consists of the timestamps AD, A, CA and C. Each line represents one log entry. A log entry is a measurement of a timestamp, including start and end time. In a log entry, the fields are separated by commas. Below a schematic log entry is shown. The detailed description of all fields can be found in table 2.1.

`id`,`iteration`,`timestamp`,`description`,`duration`,`start`,`end`

Fields	Description	Type	Example
id	id of the corresponding query	int	528
iteration	iteration of the corresponding query	int	0
timestamp	name of the measured timestamp	string	CA
description	further information	string	loadingMapData
duration	measured duration in s	float	6.51
start	start time of measurement	ISO 8601 [3]	2020-11-26T15:29:01.595Z
end	end time of measurement	ISO 8601 [3]	2020-11-26T15:29:01.595Z

Table 2.1.: Description of all fields in a log entry from the application log.

In listing 2.1 an exemplary log file is shown. The log file consists of logs generated by running a detail and a summary query one after another. The first log entry for a query is always an entry for timestamp C with description "start", in which the start time for the query is found. For the first query, this log entry can be found on line 1. Afterwards timestamps AD, A and CA are logged for each UI component that gets loaded for the query. For a detail query, there are three sets of AD, A and CA, because apart from the table UI component the client also loads the chart and the map UI component, e.g. lines 2-10 in the example. In a summary query, only a table gets displayed. Therefore only the table UI component gets loaded and so only one set of AD, A and CA can be found, e.g. lines 13-15 in the example. The UI component related to a log entry can be identified over the description field, e.g. line 3: timestamp A for the chart UI component (description = "/api/charts/hull"). For every query, the last entry in the log is again a log entry for timestamp C. The description for the end timestamp is "end". The log holds the end time of the query. In the example the last log entry for the first query can be found on line 11, stating "end" in the description and the end time.

```
1  898,0,C,start,0,2020-11-26T15:29:00.761Z
2  898,0,AD,charts/hull,2.481,2020-11-26T15:28:59.928Z,2020-11-26T15:29:02.409Z
3  898,0,A,/api/charts/hull,2.503,2020-11-26T15:28:59.920Z,2020-11-26T15:29:02.423Z
4  898,0,CA,loadingChartData,2.543,2020-11-26T15:29:01.595Z,2020-11-26T15:29:04.138Z
5  898,0,AD,map/cantons,5.696,2020-11-26T15:28:59.950Z,2020-11-26T15:29:05.646Z
6  898,0,A,/api/map/cantons,6.35,2020-11-26T15:28:59.919Z,2020-11-26T15:29:06.269Z
7  898,0,CA,loadingMapData,6.51,2020-11-26T15:29:01.595Z,2020-11-26T15:29:08.105Z
8  898,0,AD,table/paginateTD,7.034,2020-11-26T15:28:59.970Z,2020-11-26T15:29:07.004Z
9  898,0,A,/api/table/paginate,7.077,2020-11-26T15:28:59.930Z,2020-11-26T15:29:07.007Z
10 898,0,CA,loadingTableData,7.17,2020-11-26T15:29:01.595Z,2020-11-26T15:29:08.765Z
11 898,0,C,end,0,,2020-11-26T15:29:09.741Z
12 528,0,C,start,0,2020-11-26T15:49:56.823Z
13 528,0,AD,table/paginate,3.943,2020-11-26T15:49:55.341Z,2020-11-26T15:49:59.284Z
14 528,0,A,/api/table/paginate,4.042,2020-11-26T15:49:55.289Z,2020-11-26T15:49:59.331Z
15 528,0,CA,loadingTableData,4.117,2020-11-26T15:49:56.983Z,2020-11-26T15:50:01.100Z
16 528,0,C,end,0,,2020-11-26T15:50:01.114Z
```

Listing 2.1: Example for the content of an application log file.

Database Log

In the database log, each log entry can spread over multiple lines. This is due to the fact, that the whole SQL statement is logged. SQL statements in the feedbase often include line breaks. This makes it a bit harder to read the logs. A schematic log entry is shown below. The highlighted fields are the fields, which are relevant for the analysis. These fields are further described below. The description of the other fields as well as further information can be found in the PostgreSQL documentation.

```
timestamp,user,database,process_id,connection_from,session_id,session_line_num,command_tag,
    session_start_time,virtual_transaction_id,transaction_id,error_severity,sql_state_code,
    message,detail,hint,internal_query,internal_query_pos,context,query,query_pos,location,
    application_name
```

Listing 2.2: Source: PostgresQL documentation [1]

An exemplary database log entry is shown below. The field `user` stores the name of the database user, who is responsible for the query. This field is used to remove database logs generated by the system or other users. The feedbase uses the database user "php_client". All log entries issued from users other than "php_client" are ignored. The second relevant field `message` stores all the information about the SQL statement that was run. The field consists of two parts. The first part contains the duration, e.g. 3931.329 ms. The second part is the actual SQL statement which was run. For the queries that should be measured, the SQL statement always starts with the identifier of the database request, e.g. table/paginate-2 528-0.

```
2020-11-26 15:49:59.130  UTC,"php_client","tfdb",15993,"89.217.56.236:50124",5fbfbbfe.3e79
    ,2,"SELECT",2020-11-26 15:49:55 UTC,4/0,0,LOG,00000,
    "duration:  3931.329 ms statement:  - table/paginate-2 528-0
 WITH formulas AS (
 SELECT
 id_feed AS feed_key,
...
 SELECT COUNT(*)
 FROM rows AS total;",,,,,,,,,""
```

2.3.3. Log Preprocessing

Immediately after the test-run, the database logs must be copied to the application server. This can be done by calling the endpoint `GET /api/logs/database/import`. This endpoint uses sftp to copy the database logs from the database server. The credentials used for copying are defined in `params.json` by adding a key-value pair to `"db"`. The key value-pair could look like:

```
"logs": {
      "user": "userName",
      "password": "password"
    }
```

The defined user must have read access to the database logs. Alternatively, a private key can be defined instead of a password. The key for defining a private key is `"privateKey"` and the value is the path to the file that contains the key.

To preserve the logs and prevent overwriting the application and the database log are renamed right after the test-run and the import. The following schema is used for the database log: `postgresql-yyyy-mm-dd-n.csv`, where `yyyy` is the full year, `mm` is the full month, `dd` is the day of the month and `n` is the number of the test-run. The active application log is renamed according to the following schema: `stopwatch-yyyy-mm-dd-n.txt`. The variables are the same as in the naming of the database log. Renaming is done in the shell of the application with the following commands:

```
cd server/logs;
cp postgresql-yyyy-mm-dd.csv postgresql-yyyy-mm-dd-n.csv
cp stopwatch.txt stopwatch-yyyy-mm-dd-n.txt
```

Preprocessing is done every time the log data is requested. It involves several steps:

1. Filter database logs
2. Order database and application logs

3. Insert database log entries into application logs
4. Calculate differential duration

Filter database logs

The raw database log entries contain way too much information that is not needed. Additionally, there are a lot of log entries that are not relevant. In a first step, all log entries that were not made by the application tier are eliminated. The relevant log entries in the database log contain information about the duration of a statement and include the full statement itself. The comments at the beginning of the statements are used to identify all relevant database queries. The log entries are formatted and unneeded information is eliminated. Filtering is done on the raw database log file. The output is an array which contains all relevant log entries formatted according to the application log format.

Order database and application logs

Because the client tier requests the data for the chart, map and table UI components asynchronously it is not possible to predict the order in which the requests are logged. To visualize the problem an exemplary application log file is showed ordered and unordered. The log file consists of logs for two UI components (chart and map). The logical order in the application log is as follows: AD gets logged first, followed by A and CA. This is visualized in listing 2.3. AD and A are logged from the application tier, while CA is logged from the client tier. Because of that it frequently happens that CA is logged with a delay, causing it to be logged at a wrong position. This is shown in listing 2.4, where CA of the chart UI component is logged after AD and A of the map UI component.

```
898,0,C,start,0,2020-11-26T15:29:00.761Z
898,0,AD,charts/hull,2.481,2020-11-26T15:28:59.928Z,2020-11-26T15:29:02.409Z
898,0,A,/api/charts/hull,2.503,2020-11-26T15:28:59.920Z,2020-11-26T15:29:02.423Z
898,0,CA,loadingChartData,2.543,2020-11-26T15:29:01.595Z,2020-11-26T15:29:04.138Z
898,0,AD,map/cantons,5.696,2020-11-26T15:28:59.950Z,2020-11-26T15:29:05.646Z
898,0,A,/api/map/cantons,6.35,2020-11-26T15:28:59.919Z,2020-11-26T15:29:06.269Z
898,0,CA,loadingMapData,6.51,2020-11-26T15:29:01.595Z,2020-11-26T15:29:08.105Z
898,0,C,end,0,,2020-11-26T15:29:09.741Z
```

Listing 2.3: Example of an ordered application log file

```
898,0,C,start,0,2020-11-26T15:29:00.761Z
898,0,AD,charts/hull,2.481,2020-11-26T15:28:59.928Z,2020-11-26T15:29:02.409Z
898,0,A,/api/charts/hull,2.503,2020-11-26T15:28:59.920Z,2020-11-26T15:29:02.423Z
898,0,AD,map/cantons,5.696,2020-11-26T15:28:59.950Z,2020-11-26T15:29:05.646Z
898,0,A,/api/map/cantons,6.35,2020-11-26T15:28:59.919Z,2020-11-26T15:29:06.269Z
898,0,CA,loadingChartData,2.543,2020-11-26T15:29:01.595Z,2020-11-26T15:29:04.138Z
898,0,CA,loadingMapData,6.51,2020-11-26T15:29:01.595Z,2020-11-26T15:29:08.105Z
898,0,C,end,0,,2020-11-26T15:29:09.741Z
```

Listing 2.4: Example of an unordered application log file

To achieve a correct ordering and for easier access when inserting the database log entries into the application log, the log entries of the database and application log are restructured. The input of the function responsible for restructuring is an array of log entries in the application log format. Each log entry is split into individual fields already. An example using the logs from above is shown in listing 2.5. Only the first four rows of the logs above are shown, the other rows follow the same principle.

```
[
 [898,0,"C","start",0,"2020-11-26T15:29:00.761Z"],
 [898,0,"AD","charts/hull",2.481,"2020-11-26T15:28:59.928Z","2020-11-26T15:29:02.409Z"],
 [898,0,"A","/api/charts/hull",2.503,"2020-11-26T15:28:59.920Z","2020-11-26T15:29:02.423Z"],
 [898,0,"AD","map/cantons",5.696,"2020-11-26T15:28:59.950Z","2020-11-26T15:29:05.646Z"],
  ...
]
```

Listing 2.5: Example of a correctly formatted log array used as input for restructuring

The desired output is a JavaScript Object. It is a key-value list of all query iterations. The key consists of the queryId and the iteration. The value is a JavaScript Object consisting of an array of log entries for each UI component. Additionally, there is an array with the start and end log entry for C because it is not possible to map C to one specific UI component. Each `logEntry` is an array, containing the fields of the log entry. An exemplary output using the input from above is visualized in listing 2.6. The log entries are shortened for better visualization.

```
{
  "898-0": {
    "map": [
      [898,0," AD "," map/cantons ",5.696,...],
      [898,0," A "," /api/map/cantons ",6.35,...],
      [898,0," CA "," loadingMapData ",6.51,...],
    ],
    "chart": [
      [898,0," AD "," charts/hull ",2.481,...],
      [898,0," A "," /api/charts/hull ",2.503,...],
      [898,0," CA "," loadingChartData ",2.543,...],
    ],
    "table": [],
    "C": [
      [898,0,"C","start",0,"2020-11-26T15:29:00.761Z"],
      [898,0,"C","end",0,,"2020-11-26T15:29:09.741Z"],
    ]
  }
}
```

Listing 2.6: Restructured log entries from listing 2.4

Insert database log entries into application logs

With the ordered application and database log everything is ready to insert the database log entries into the application log. To do that, the system goes through all query iterations. For each UI component in a query iteration the system searches for a corresponding log entry in the database log. If there are multiple database log entries for one UI component (table component) the log entry having the maximal duration of all log entries for this query iteration

and UI component is taken. The selected database log entry gets inserted into the application log.

Calculate differential duration

The differential duration defines the difference between the whole duration of a timestamp and the time of the next inner timestamp, e.g. CA has a duration of 20 seconds and A has a duration of 8 seconds. Thus the differential duration of CA is: $20s - 8s = 12s$. Following subtractions are done for the differential duration: (D, AD, ... represent the duration of the corresponding timestamp, which is calculated by subtracting the start time from the end time)

- $D_{diff} = D - 0$
- $AD_{diff} = AD - D_{max}$
- $AD_{diff} = A - AD$
- $CA_{diff} = CA - A$
- $C_{diff} = C - CA_{max}$

Visualization

The collected timestamps get visualized in Microsoft Excel. The excel-document automatically gets the data from different endpoints and visualizes them. The endpoints use the preprocessing described above and then do aggregation operations so that MS Excel only needs to do the visualization of the data. All durations visualized in Microsoft Excel are calculated as an arithmetic mean of three measurements taken one after another. The used endpoints are:

- `GET /api/logs/{{fileName}}/plots`
  for visualizing the performance of system components.
- `GET /api/logs/{{fileName}}/plots/{{timestamp}}`
  for visualizing the performance of a timestamp for different UI components.

`{{fileName}}` is used to set the name of the log file(s) to be analysed. The application and database log filename can be specified individually, separated by a comma, starting with the application log filename. `{{timestamp}}` is used to set the timestamp to be analysed.

2.4. Database Optimization

In general, the process of optimization in a database is done in four different steps. First, an overall picture of the database performance is measured. This first look can give an indicator where the performance problem lies. If specific queries show a slow performance while the rest of the system performs fine, it is very probable that the problem lies in the specific queries. When the whole system has slow performance, the source of the problem can be more diverse.

The second step involves a detailed analysis of the problematic parts. Inefficient queries are analysed with the help of the query plans generated by the optimizer. The goal is to find out where the problem is located exactly and to find a solution. In the third step the found solution is implemented. After the optimization, the query gets analysed again to see if the improvement achieved its target. If the outcome is not satisfying steps two to four can be redone, i.e. do another analysis, optimization, and evaluation.

To locate and optimize SQL statements it is important to have a profound understanding of performance-relevant elements in a database system. In the next sections, the focus is on describing the elements relevant for the optimization of the SQL statements in the feedbase. Thus the further sections are written specifically for PostgreSQL databases.

2.4.1. Query Plans

PostgreSQL, like any other database system, creates different plans on how to execute a query. Each plan has estimated costs, which are calculated based on the database statistics. The database system then chooses the "optimal" query plan based on the estimations [4]. If the estimates in the query plans are very different from the reality, the database can choose a non-optimal plan. The query plan is crucial when it comes to understanding performance problems because it contains detailed information about the query. This helps to find out which operation was costly and how the query could be optimized. A query plan has the structure of a tree, where the nodes are different actions a database system can perform (sort, aggregate, scan, join, ...) [4]. The leaf nodes of the tree (nodes at the bottom level) are scan nodes, which return raw rows from a table [4]. In PostgreSQL, the query plan is output with the `EXPLAIN` statement. In such a query plan there is one line for each node in the tree showing the type of the node and the cost estimates the query planner made for the execution of the node [4]. The cost estimates are in an arbitrary unit and for example, cannot be transformed into seconds. There can be additional lines for a node with more properties of the node [4]. These additional pieces of information are indented from the node's first line. Child nodes start with an arrow ("`->`") and are indented as well.

The `EXPLAIN` statement returns the query plan but doesn't actually run the query. When using `EXPLAIN ANALYZE` the query is actually executed and PostgreSQL provides an enriched query plan, that contains information about the execution time and the actual amount of rows output by a node. This is extremely helpful to find nodes that the query planner over- / underestimated.

2.4.2. Finding the Bottle-Neck in a Query Plan

We strongly suggest using `EXPLAIN ANALYZE`, which yields an enriched query plan that is still readable. Most of the bottle-necks can be easily spotted by going through the query plan line by line. The easiest way to read a query plan is to start at the innermost node because this gets executed first. Inside a node, the cost estimation is the first thing to look at. The focus is on big increases in costs compared to the previous (more inner) node. When a big increase is found, the next step is to look at the actual execution time of the node. Is there a big increase as well? Do the costs of that node make up the biggest part of the total costs? If yes, this node

is probably the bottle-neck. The second thing to look at is the estimated and actual row count. In practice, these counts hardly ever are exactly the same. If the actual row count is way off, i.e. 100x / 1000x more than the estimated row count, the query planner did a bad estimation. This can eventually influence the query planner to choose a non-optimal plan.

```
1  GroupAggregate (cost=32465.41..32467.77 rows=1 width=48) [7]
2    Group Key: points.id, points.an_id, points.geom_number
3    -> Sort (cost=32465.41..32465.41 rows=1 width=28) [6]
4      Sort Key: points.id, points.an_id, points.geom_number
5      -> Merge Join (cost=1749.44..32465.40 rows=1 width=28) [5]
6        Merge Cond: ((points.id = stats.id) AND (points.an_id = stats.an_id))
7        Join Filter: st_equals(points.dp, st_makepoint(stats.day_normalized, stats.
             quantity_normalized))
8        -> Sort (cost=212.98..219.93 rows=2782 width=48) [4]
9          Sort Key: points.id, points.an_id
10         -> Seq Scan on geometrypoints points (cost=0.00..53.82 rows=2782 width=48 [3]
11       -> Sort  (cost=1536.46..1579.96 rows=17399 width=36) [2]
12         Sort Key: stats.id, stats.an_id
13         -> Seq Scan on statsnormalized stats (cost=0.00..310.99 rows=17399 width=36) [1]
```

Listing 2.7: Example of a query plan with a bottle-neck created with EXPLAIN

In listing 2.7 an example of a query plan with a bottle-neck is shown. For each row that is read, the costs are analysed. In the query plan above the startup costs of a node are highlighted in green, while the total costs are highlighted in blue. In the start-up costs the costs of the child nodes are included. The total costs of the query is equal to the total costs of the first line (32467.77 in the example). For an easier understanding of how to read such a query plan, numbers in [], highlighted in orange, are displayed. They show the order in which the query plan is read. The first row which is read is a scan node with a total cost of 310.99. The next line is a sort node with 1579.96 total costs. The two sort nodes ([2] and [4]) are located on the same level and have the same parent node. This implies that the costs of the parent only include the costs of the largest child node. Node [4] is the last child node of node [5] that gets visited. So the next node to be analysed is node [5]. One proceeds in this fashion until a large increase in costs is discovered. The large increase is of course always relative to the total costs of the query (32467.77). In this query plan, a large increase can be seen in line 5, which is read as the fifth line. The startup costs of 1749.44 include 1579.96 from the child node with the biggest costs. The startup costs are thus very small for this node. However the total costs (32465.40) are very large. They are almost as high as the total costs. This shows, that the current node is the bottle-neck in this query.

2.4.3. PostgreSQL's Performance Enhancing Features

PostgreSQL offers a lot of features that can enhance performance drastically. In this section Just-in-Time Compilation and Parallel Queries are presented.

Just-in-Time (JIT) compilation

Just-in-Time (JIT) compilation was introduced with PostgreSQL version 11. It is the process of turning some form of interpreted program evaluation into a native program and doing so at

run time [5]. For example, instead of using general-purpose code that can evaluate arbitrary SQL expressions to evaluate a particular SQL predicate like `WHERE a.col = 3`, it is possible to generate a function that is specific to that expression and can be natively executed by the CPU, yielding a speed-up [5]. For the expression `WHERE a.col = 3` the result would be a function, written in a low-level programming language, that evaluates the expression using background information on this specific expression, e.g. the data type stored in column `a.col` and the specific condition of the expression `= 3`. JIT compilation is like having a software engineer sitting between the database user and the database [6]. For each query the database user issues, the software engineer writes a specific program (in C++ / Java ...) and hands it to the database [6]. The database then executes this specialised program, which is faster than a general solution. To write such a specific program, information about the data schema, the structure of the query, and the data types is needed [6]. In PostgreSQL expression evaluation and tuple deforming are accelerated by using JIT compilation [5]. Expression evaluation is the process of evaluating aggregates, projections, `WHERE` clauses, and target lists [5]. Tuple deforming is the procedure of transforming a tuple from the disk to its in-memory representation [5].

JIT compilation only makes sense for complicated queries with lots of input tables and expressions, because in that case, the acceleration is larger than the additional overhead needed to do the JIT compilation. In this thesis, many expressions profited from the JIT compilation. Therefore it is not possible to narrow down the usage of JIT compilation to specific expressions, that profit a lot.

The summary of a query plan has a section for the JIT compilation, whenever it was used in the SQL expression that was run.

Parallel Queries

PostgreSQL version 9.6 introduced support for parallel queries. This support has been improved over time, especially with version 11 adding even more functionality [7]. The idea behind parallel queries is simple: CPU-intensive tasks are split so that multiple workers can work on the same task in parallel. At the time this thesis was written, sequential scans, index scans (btrees only), bitmap heap scans, joins (all types of joins), btree creation (CREATE INDEX), aggregation, and append can be done in parallel [7]. By default, the database system automatically uses parallel queries, whenever it is possible and the use of parallel queries potentially yields a speed-up. One can only configure how many parallel workers the database system can use at maximum. The corresponding property in the configuration is called `max_parallel_workers_per_gather`. In listing 2.8 an exemplary query plan of a join using a parallel feature (parallel scan) is shown. Parallel queries are easily identified in the query plan by the "Gather" node, highlighted in yellow. Every parallel query has this node to collect all the data from the workers and do a final aggregation on it. In orange additional information about the parallel execution is shown: the use of two workers is planned. In green, the parallel node is highlighted, in this case, a parallel sequential scan.

```
1 -> Gather (cost=1000.57..54554.18 rows=6 width=4)
2   Workers Planned: 2
3   ->  Nested Loop (cost=0.57..53553.58 rows=2 width=4)
4     -> Parallel Seq Scan on fact_table fact_table_1 (cost=0.00..53521.71 rows=2 width=12)
5       Filter: ((lims_number)::text <> '0-const'::text) AND (id_nutrient_fkey = 112)
6     -> Index Only Scan using timekey_idx on d_time d_time_1 (cost=0.29..8.30 rows=1 width=4)
7       Index Cond: (time_key = fact_table_clean_1.id_time_fkey)
```

Listing 2.8: Excerpt of a query plan using parallel features, created with EXPLAIN

2.4.4. Spatial Indexes

Spatial indexing is a key part of spatial databases. Without indexing, every search would need to do a sequential scan of every record in the database. With a spatial index, a search can be accelerated by offering a tree structure to access a particular record quickly [8]. Especially joins can benefit a lot from spatial indexes. When joining two tables with 10'000 records each, without indexes PostGIS would require 100'000'000 comparisons. With the right indexes, the costs can drop as low as 20'000 comparisons [8].

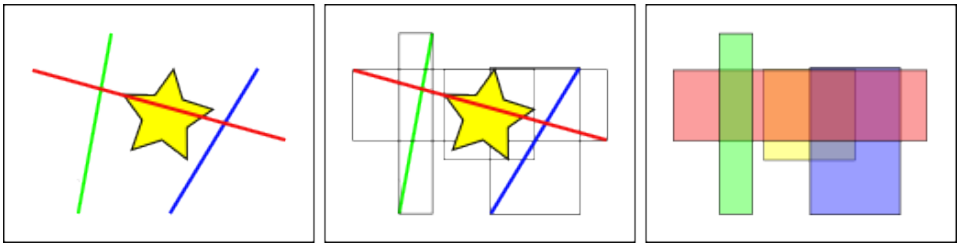


Figure 2.4.: Illustration to show the construction of a spatial index, source: [8]

In figure 2.4 the construction of a spatial index is shown. The left image shows the geometric features, the middle image shows the features with their bounding boxes and the right image shows the bounding boxes only. In contrast to standard database indexes that build a hierarchical tree based on the values of the records, spatial indexes don't build a tree based on the geometric features themselves, but on the bounding boxes of these features. This leads to a new problem. In the left image of figure 2.4 only one line intersects the star: the red line. When looking at the bounding boxes, there are two features intersecting the yellow box: the red and the blue box. The blue box is thus a false positive. The solution is a "two pass" system. In the figure above the first pass would look at all bounding boxes and find the ones that intersect the yellow box: the red and the blue box. In the second pass, the goal is to eliminate the false positives. Therefore the geometric shape of the features is compared instead of the bounding box, resulting in finding only the red line intersecting the yellow star. What speeds up a database system is the first pass, reducing the number of needed calculations radically by first evaluating the approximate index [8].

In PostgreSQL (with the PostGIS extension) a spatial index is defined with the following command: `CREATE INDEX "<index_name>"ON <relation_name> USING GIST (<attribute_name>);`. `<index_name>` is the name of the index to be created. `<attribute_name>` is the name of the spatial attribute for which the index is created in the relation defined by its name in `<relation_name>`.

2.4.5. Spatial Joins

A spatial join is a join of two spatial features according to some predicate that makes use of the spatial attribute values [9]. Simple examples for spatial joins are:

- "Combine all mountain tops with the canton they belong to."
  `mountain_tops join cantons on (point inside area)`
  `mountain_tops` and `cantons` are spatial features, that are joined with the predicate `inside` using the spatial attributes `point` and `area`.

- For each cantonal border find the mountain tops within less than 500 meters.
  `mountain_tops join cantonal_borders on (distance(point,route) < 500)`
  `mountain_tops` and `cantonal_borders` are spatial features, that are joined with the predicate `distance(attribute1, attribute2) < 500` using the spatial attributes `point` and `route`.

(The examples were inspired by Güting [9])

Because spatial joining is a crucial operation, it is important that it is supported by spatial indexing [9]. The precondition for a spatial join to use spatial indexes is, that at least one of the operands in the join is represented in a spatial index.

If one operand of the spatial join has a spatial index, an index join can be used [9]. An index join is a classic technique, that usually get used with B-tree indexes [9]. This strategy can be used with spatial index structures as well [9]. The basic strategy of an index join is the same as in a nested loop join. It uses two loops that are nested. The outer loop goes through all rows in one table. The inner loop goes through all rows in the other table, for each of the outer rows [7]. When comparing the row from the outer table with the rows from the inner table, an attribute from the outer row can be used to search in the index of the inner table. This is the difference between an index join and a nested loop join. For a spatial join, the inner table has a spatial index on the joined attribute.

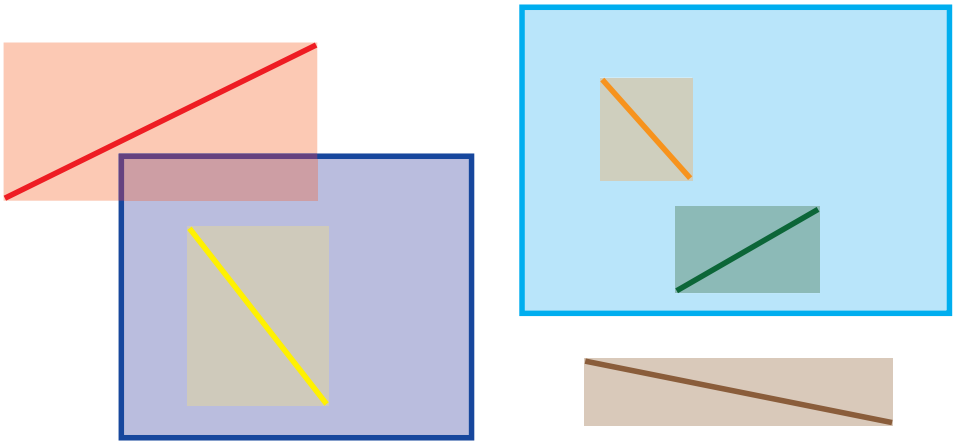


Figure 2.5.: Illustration to show a spatial join between rectangles and lines

Let's look at an example of an index join. In figure 2.5 rectangles (dark and light blue) and lines (yellow, orange, red, green) are shown with their bounding boxes (faded colour of

the feature). The rectangles are stored in a table called `rectangles`, the lines are stored in a table called `lines`. Each table has a spatial index (using the bounding boxes). The following query needs to be executed: For each rectangle, find the intersecting lines. More formally: `rectangles join lines on (intersect(lines,rectangles))`. Because `rectangles` is the smaller table, it is chosen as the outer table. The database system loops through every rectangle and uses its bounding box to search for intersections in the index of `lines`. For the dark blue rectangle, it finds the red and the yellow line in the index. For the light blue rectangle, the orange and the green line are found. Even though the bounding box of the red line intersects the bounding box of the dark blue rectangle, the line and the rectangle are not intersecting in reality. This is a false positive and is removed in a second pass (as described in section 2.4.4). Thus in the example, for each rectangles, two exact comparisons with lines have to be done.

If there was no spatial index on the tables in the example above, the database system would need to check every line against every rectangle. In our example for both rectangles, five exact comparisons would need to be done. This yields a total of 10 exact comparisons without an index and only 4 exact comparison, when an index can be used. With a growing number of objects, this can lead to bad performance when not using an index.

2.4.6. Common Table Expression (CTE) vs. Temporary Table

Common Table Expression

Common table expressions (CTEs), also known as the WITH clause, are commonly used to make queries more readable. They are a nice way to execute things only once in a SQL statement while reusing the result of it multiple times throughout the query [7].

The syntax for using a CTE in a statement is: `WITH <cte name> AS (<selectcommand>) <statement>`. In a basic example, the CTE `cte1` is created as the join of two tables `table1` and `table2`. Then everything from the CTE `cte1` is selected. In PostgreSQL this would look like this:

```
WITH cte1 AS (SELECT * from table1 JOIN table2 on (table1.id = table2.id)) SELECT * FROM cte1;
```

In PostgreSQL versions pre 12, CTEs come with some drawbacks. First, the optimizer is not able to inline the query to turn it into something faster. Secondly, the result of a CTE is always calculated, like for an independent query. These two specialties hinder the optimizer to do smart things and therefore CTEs can become optimization fences. Since PostgreSQL 12 the shortcomings in the optimizer are not a problem anymore, because version 12 has more options to deal with CTEs smartly [7].

Temporary Table

Another option to structure queries is the use of temporary tables. In PostgreSQL a temporary table can be created by adding the keyword `TEMPORARY` to the `CREATE TABLE <tablename>` statement, yielding: `CREATE TEMPORARY TABLE <tablename>`. Per default temporary tables are usable within the same session and are dropped when the session ends. There is an option to drop the table when a transaction is ended. This option can be enabled by appending `ON COMMIT DROP` to the `CREATE TEMPORARY TABLE <tablename>` statement [10]. Temporary tables can have indexes and dedicated statistics like a normal table. Temporary tables are not analysed or vacuumed by the

PostgreSQL autovacuum daemon. When issuing complex queries it is therefore advisable to run `ANALYZE` on the temporary table [10].

Comparison

A CTE is only usable for the statement that created the CTE. A temporary table is usable during a whole session or transaction. An advantage of using temporary tables is the ability to add indexes to them. This is not possible with CTEs. Creating accurate statistics with `ANALYZE` is another benefit of temporary tables over CTEs. The downside of temporary tables is the cost of initializing them. These costs are higher than for a CTE. Finally, CTEs are more readable because the command to create a CTE is much shorter. In this thesis, it could be observed that CTEs generally are less performant than temporary tables for large data sets. In section 5.1 the duration of a query was decreased from 4.2s to 2s only by replacing CTEs with temporary tables.

Replace a CTE with a Temporary Table

To replace a CTE with a temporary table the scope of a temporary table must be similar to the scope of a CTE. Per default temporary tables are available for the full session. It is possible to change the scope of the temporary table to only the current transaction (by adding `ON COMMIT DROP` as seen above). With this change, the scope of a temporary table is roughly the same as the scope of a CTE. PostgreSQL offers a shorthand to create a temporary table and fill it with data computed by a `SELECT` command - all in one statement: `CREATE TABLE <tablename> AS <selectcommand>` [11]. Summarizing the findings, a CTE can be replaced by a temporary table using the following command:
`CREATE TEMPORARY TABLE <tablename> ON COMMIT DROP AS <selectcommand>`.

3. Bottle-Neck Assessment

3.1. Timestamp Analysis

In figure 3.1 the mean of the timestamps in the three test-runs is shown. D has the highest means in all test-runs. C is larger than AD, A, and CA, while A is only slightly higher than AD and CA. The underlying data for figure 3.1 can be found in table B.1 in the appendix. The means suggest that the bottle-neck can be found in timestamp D.

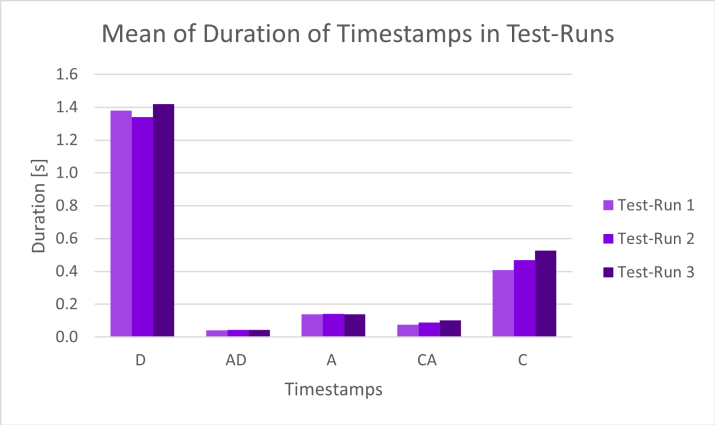


Figure 3.1.: Mean of the differential duration for all test-runs

Test-run	C	CA	A	AD	D
1	6	1	0	0	30
2	11	0	0	0	26
3	9	1	0	0	27

Table 3.1.: Amount of times a timestamp had the longest duration in a query

In Table 3.1 it is shown that D often has the longest duration. C also takes the longest in some queries. CA rarely takes the longest. A as well as AD never take the longest.

The visualizations of all results can be found in the appendix. During testing, some observations showed that timestamp C can be affected strongly by the client device. There were increases of around 2s in timestamp C with the same setup on the server and the database, e.g. query 794 in test-run 1. It is assumed that these large increases are due to some kind of tasks that the client device ran in the background. For D there are some fluctuations as well, e.g. query 153 in test-run 1 is higher than in test-run 2. An increase of 17s in D for a single

iteration in a test-run was observed. This major increase could be the result of a database downtime or some other influence, that is not controllable. Apart from the above-described differences, the results of the three test-runs look quite similar overall. A, AD, and D are quite stable throughout the test-runs, while C varies more. CA is low in all test-runs over all queries.

3.2. Client UI Component Analysis

In the next step, it is investigated how the different client UI components (map, chart, and table) are compared to each other. The client tier requests the information of these three UI components in parallel and so the client tier can only show the whole user interface when all UI components received their information. The biggest durations for CA and AD are very low (around 0.6 seconds for CA and 0.5 seconds for AD in test-run 2). Because of this CA and AD are not analysed further.

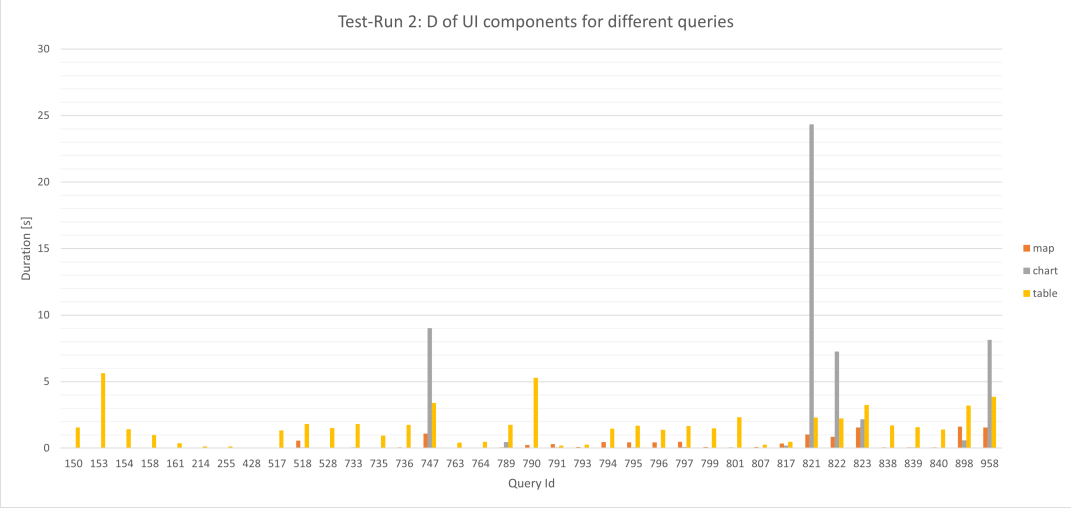


Figure 3.2.: Test-run 2: Duration of timestamp D for UI components for different queries

For timestamp D the results are consistent over all three test-runs. The results from test-run 2 are shown in figure 3.2. Chart queries can take very long but do so in only a few cases. Four chart queries (747, 821, 822, 958) take between 7 and 24 seconds to complete. Table queries take between 1 and 3 seconds in many cases. In queries 150 and 790 the table UI component takes up to 5-6 seconds. Map queries mostly have durations shorter than a second. Only in few cases, the duration rises slightly over one second, e.g. queries 823 and 898.

Besides timestamp D, timestamp A showed an interesting pattern as well. In figure 3.3 the durations of timestamp A for the different client components are visualized. The duration of timestamp A is very low for the chart and table UI component. The map UI component has some queries where the duration is over 1 second, e.g. queries 821 and 822.

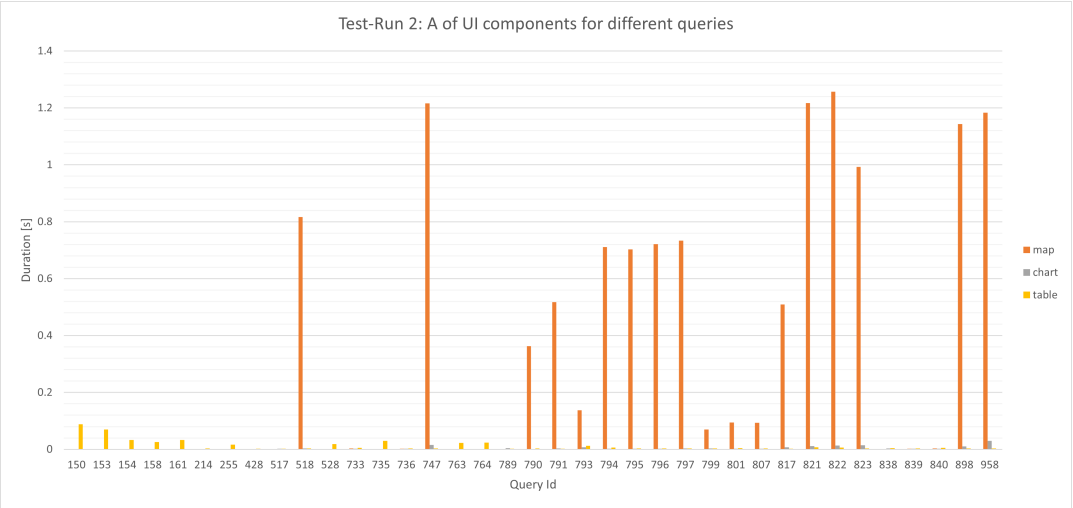


Figure 3.3.: Test-Run 2: Duration of timestamp A for UI components for different queries

3.3. Summary

Based on table 3.1 especially D and in some cases C can take a long time. In figure 3.1 it is shown that D has big means, while AD, A, and CA have significantly lower means.

The table query sometimes has long durations in D. The chart query has very long durations in D in few cases. The map component never takes longer than 1.5 seconds in D and therefore is not that relevant for the very bad performance of the feedbase with some queries. Nevertheless timestamp A and D in the map component should also be considered if the performance of each component should drop clearly below one second.

The results of the measurements suggest investigating D for the UI components chart and table because they have a major impact on the performance. C has slightly higher means than AD, A, and CA in figure 3.1, which suggests to also analyse C closer. Because C depends heavily on the hardware of the client it would be interesting to look at C on other client devices.

The hypothesis stated in the introduction can be approved partially. Based on the measurements the main limiting part is the PostgreSQL database. Additionally, the client tier is responsible for bad performance in some cases as well.

3.4. Validation of Analysis

Two validation-runs are done to confirm the method of measurement and the findings in the analysis above. The validation-runs are executed on six selected queries listed in table 3.2.

3.4.1. Validation-Run 1: Blocking Code

In the first validation-run blocking code is added to different parts of the system. Blocking code describes code that delays the system by a specific amount of time. The duration of the

Query ID	Query Type	Query Name
153	summary	Nährstoffprofil des Futterkatalogs
428	summary	neue Fettsäureanalytik_inArbeit
791	detail	Rapssaat standard, RL, RLGC, RP
799	detail	Futterrübe, Faserfraktionen
821	detail	regionaler NEL-Gehalt in Dürrfutter 2005-2017
823	detail	Berheu > 1000 m, 2005-2017

Table 3.2.: Queries selected for the validation-runs

blocking code is set to 5 seconds delay. 5 seconds is a noticeable delay, while the duration of a validation-run is not affected much. This is important for the representativeness of a validation-run. It should have roughly the same duration as a normal test-run, so that natural fluctuations, i.e. in the network, are in the same order of magnitude. The system was extended so that each UI component (map, chart, table) could be blocked in three different timestamps (A, AD, D). The blocking can be activated by adding a query parameter to the URL (e.g. blocking the map in timestamp AD is achieved by appending `&block=mapAD` to the URL).

The results in validation-run 1 support the method of measurement. In CA fluctuations are observable in every measurement. Since CA has very small durations, these fluctuations are negligible. When blocking different timestamps in the chart UI component an increase in duration only for the blocked timestamp and no significant change of other timestamps was observed. Also, the increase in duration for the blocked timestamp is always around 5 seconds, which is the delay the blocking code added. The same could be observed for blocking different timestamps in the table UI component. When blocking A and AD in the map UI component the observations for chart and table UI components could be confirmed. The results for blocking D in the map UI component are shown in figure 3.4. On the left side, the reference measurements for AD and D are displayed. On the right side, the measurements of AD and D having blocking code in D for the map component are shown. When comparing the durations of the map component in D (the lower plots) one can see that blocking the map component in D increases the duration of D for the map component as expected. Interestingly an increase in AD for all components (the upper plots) can be observed as well.

For further analysis of the increase in AD seen while blocking D in the map UI component, this specific blocking was run another two times. In both measurements, AD showed no increase while D increased by 5s as expected. Because other components also showed an increase in AD in the initial measurement it is probable that the network connection between A and D was slow at the time the measurement was made.

3.4.2. Validation-Run 2: Cache database results

In the second validation-run the database results are cached. Caching the database results is done not only to support the method of measurement but to also have an impression of how the performance of a perfectly optimized database would look like (durations around 0 seconds). The database results are cached in json files on the application tier. Mapping the request to the

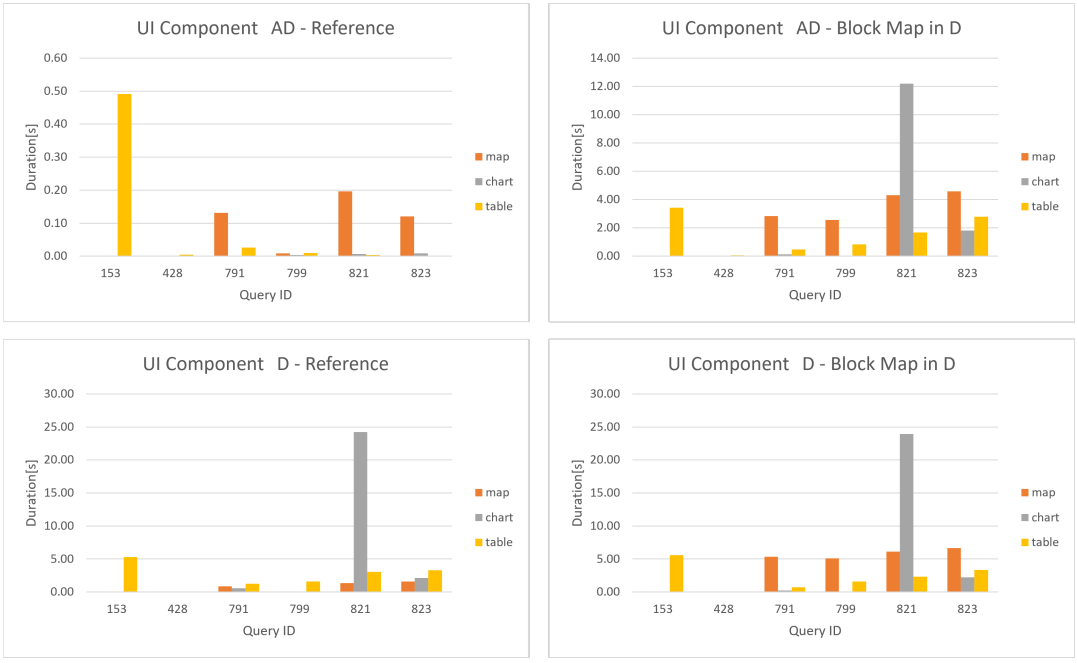
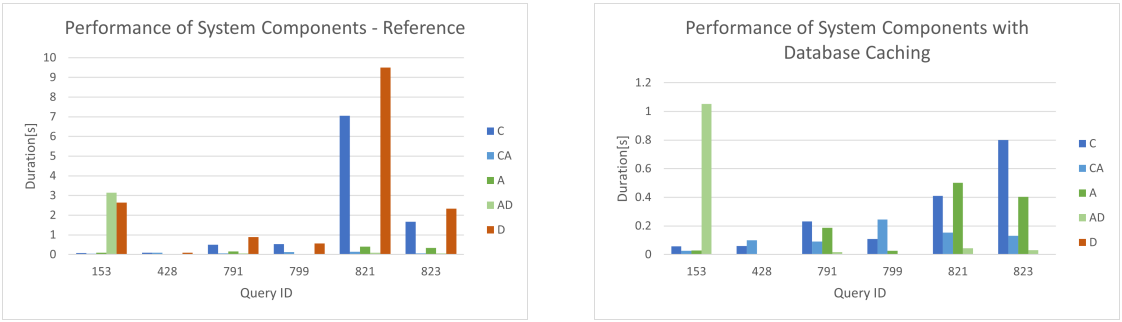


Figure 3.4.: Validation-Run 1: Block Map in D

cached files is done over the hash of the request body.

On the right-hand side of figure 3.5 timestamp D is zero for all queries. AD is not zero because it includes the time the application tier needs to access and read the json-file. In the case of query 153, the json-file is very large and therefore there is a long duration in AD. Timestamps C, CA, and A perform in the same order of magnitude. This validation-run shows, that it can be very effective to optimize D. The timestamps in validation-run 2 rarely have durations bigger than a second which results in a good performance. Of course, D cannot be minimized by the amount they were minimized in this validation-run. Nevertheless, a noticeable improvement of the performance is possible.



(a) Reference Measurement

(b) Measurement with Database Caching

Figure 3.5.: Validation-Run 2: Performance Measurement with Database Caching

4. Chart UI Component Optimization

For the optimization, the SQL statement generated by the application for the chart query in query 821 was used because in the performance analysis this query had long durations. The full query can be found in listing C.1. The query uses 7 CTEs. To see how the execution time is distributed over the CTEs and the final select statement, the CTEs were substituted with temporary tables. This allows measuring each part separately. In figure 4.1 the duration of every part is shown. One can see that query part 7 needs optimization. It takes more than 20s. Additionally, query parts 1 and 4 are candidates for optimization.

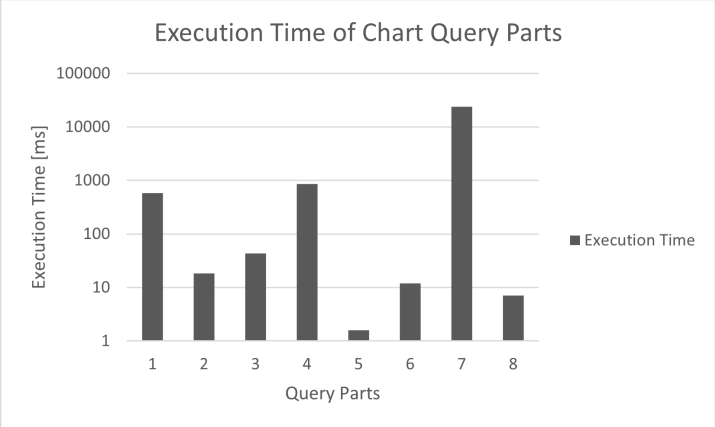


Figure 4.1.: Execution Time of Chart Query Parts displayed with a logarithmic scale.

The query plan for part 7 is shown in figure 4.1. In line 1 the total costs are highlighted in green. Going through the plan node by node, the node on line 6 stands out. The costs of this join node are highlighted in orange. The node has 1749.44 startup costs, which include 1579.97 costs of the largest child (sort node in line 14). This yields a pretty small startup cost of $1749.44 - 1579.97 = 169.48$ for the join only. The total costs of the join node are very high, 32465, and only marginally smaller than the total costs of the query: 32467.77. Therefore the join node in line 6 could be identified as the bottle-neck. To better understand what the costly node does, one has to look at the corresponding SQL statement. It can be identified by the additional information of the node. Lines 7 and 8 in the query plan state the join condition over which one can find the corresponding part in the SQL statement.

The correlated part is shown in listing 4.2. The join uses a join condition including a non-spatial (blue) and a spatial (brown) part. In his thesis, Valentin Weiss states that PostGIS features, originally intended to be used with geographical data, can be used for any data having a 2D/3D extent [12]. PostGIS allows the use of powerful clustering and grouping functions needed to process the data for the scatter plot visualization [12]. This explains why there is a

spatial part in the join condition. A 2D space is used to compute the clustered points of the scatter plot. In the first step, the tables are joined on the non-spatial join conditions (line 7 in listing 4.1). In a second step, the results of the join are filtered by the spatial join condition (line 8 in the query plan). Around 42 million rows are removed by the spatial join condition (highlighted in yellow). This explains the very bad performance of the query. The database system first computes a join resulting in around 42 million rows and then throws away most of the rows to finally output only 1544 rows. The database system chooses such a bad plan because there is no spatial index on the spatial attributes in the join condition. As described in section 2.4.5 a database system is only able to efficiently evaluate spatial joins when at least one spatial attribute has a spatial index. Because there is no spatial index, the predicate `st_equals` is evaluated after joining the rows on the non-spatial join condition.

```
1  GroupAggregate  (cost=32465.41..32467.77 rows=1 width=48) (actual time=23925.167..23928.204
       rows=788 loops=1)
2    Group Key: points.id, points.an_id, points.geom_number
3    ->  Sort  (cost=32465.41..32465.41 rows=1 width=28) (actual time=23925.128..23925.208 rows
         =1544 loops=1)
4      Sort Key: points.id, points.an_id, points.geom_number
5      Sort Method: quicksort  Memory: 169kB
6      ->  Merge Join  (cost=1749.44..32465.40 rows=1 width=28) (actual time=6.186..23920.335
           rows=1544 loops=1)
7        Merge Cond: ((points.id = stats.id) AND (points.an_id = stats.an_id))
8        Join Filter: st_equals(points.dp, st_makepoint(stats.day_normalized, stats.
             quantity_normalized))
9        Rows Removed by Join Filter:  42250144
10       ->  Sort  (cost=212.98..219.93 rows=2782 width=48) (actual time=0.695..2.277 rows=2588
             loops=1)
11         Sort Key: points.id, points.an_id
12         Sort Method: quicksort  Memory: 328kB
13           ->  Seq Scan on temp_geometrypoints points  (cost=0.00..53.82 rows=2782 width=48) (
                 actual time=0.008..0.358 rows=2588 loops=1)
14       ->  Sort  (cost=1536.46..1579.96 rows=17399 width=36) (actual time=4.845..2697.372
             rows=42249101 loops=1)
15         Sort Key: stats.id, stats.an_id
16         Sort Method: quicksort  Memory: 1660kB
17           ->  Seq Scan on temp_statsnormalized stats  (cost=0.00..310.99 rows=17399 width=36)
                  (actual time=0.007..2.468 rows=16326 loops=1)
18 Planning Time: 0.219 ms
19 Execution Time: 23928.290 ms
```

Listing 4.1: Query Plan of the join leading to slow performance

```
100 SELECT geom_number,
101   points.id AS id,
102   points.an_id AS an_id,
103   ST_MakePoint(EXTRACT(EPOCH FROM day - to_timestamp(0)) * 1000, avg_quantity) AS points
104   FROM temp_geometryPoints AS points, temp_statsNormalized AS stats
105   WHERE points.id = stats.id
106   AND points.an_id = stats.an_id
107   AND ST_Equals(dp, ST_MakePoint(day_normalized, quantity_normalized));
```

Listing 4.2: Subquery of part 7 responsible for the long duration

To improve the performance of the problematic join, spatial indexes on both spatial attributes in the `st_equals` predicate were created. Even though theoretically only one relation (typically the larger relation) needs a spatial index to enhance join performance, indexes were

created for both relations. The reason for this is that in the future the smaller table could be bigger than the larger one. With two indexes it is possible to join efficiently even in this case. A spatial index was added on the attribute `dp` in the temporary table `temp_geometryPoints` with the following command:

```
CREATE INDEX "_I_points_dp" ON temp_geometryPoints USING GIST (dp);
```

To provide a spatial index on the attribute `ST_MakePoint(day_normalized, quantity_normalized)` a new temporary table that includes this attribute was created. The index was then created on the new temporary table. Following commands were used to create the temporary table and its spatial index:

```
CREATE TEMP TABLE temp_statsNormalizedGeom ON COMMIT DROP AS
SELECT
id,
an_id,
avg_quantity,
day,
ST_MakePoint(day_normalized, quantity_normalized) as stat_geom
FROM temp_statsNormalized;
CREATE INDEX "_I_stats_geom" ON temp_statsNormalizedGeom USING GIST (stat_geom);
```

The database system needs updated statistics so that it can choose an optimal query plan using the newly created indexes. Therefore the `ANALYZE` command is issued after the indexes are created using:

```
ANALYZE temp_geometryPoints;
ANALYZE temp_statsNormalizedGeom;
```

With this optimization, the duration of part 7 drops from almost 24 seconds to nearly a second. The query plan of the optimized query (listing 4.3) shows the reason. The performance improves so much because the database system can use spatial indexes. The details on how spatial indexes help speed up a spatial join are described in section 2.4.5. Highlighted in orange the query planner chose to do the scan on the inner table using the new spatial index on `temp_geometryPoints`. As described in section 2.4.4 the lookup in a spatial index is a two-step process, because the spatial index can generate false positive matches. In the query plan, these two steps can be observed. The first step is highlighted in blue: doing the lookup in the index with an index condition. The second step is highlighted in green: filtering the results by evaluating the spatial condition exactly to eliminate possible false positives.

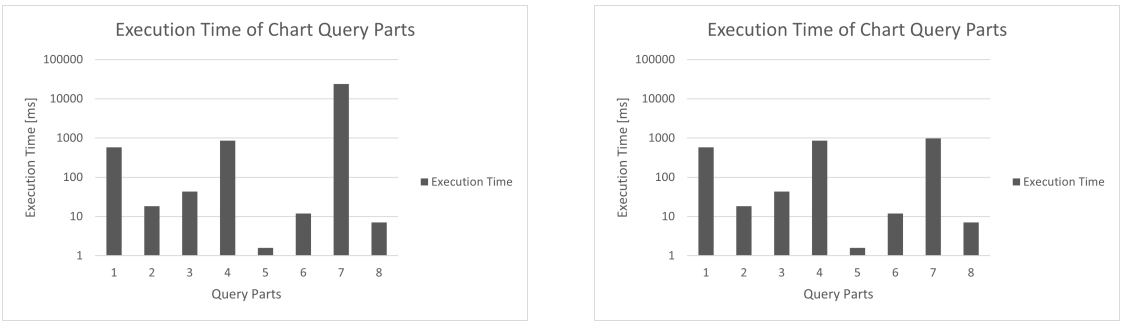
The total duration decreased but the total costs (yellow) are much higher than in the original query plan. It is assumed that this unexpected increase comes from the overestimation of the number of rows returned from the join (red). The estimated row count is much higher than the actual row count (brown).

```
HashAggregate  (cost=1591579.15.. 1592189.85 rows=788 width=48) (actual time
    =972.072..973.930 rows=788 loops=1)
  Group Key: points.id, points.an_id, points.geom_number
  ->  Nested Loop  (cost=0.14..1228594.80 rows= 229374 width=28) (actual time
      =386.692..964.072 rows= 1544 loops=1)
        ->  Seq Scan on temp_statsnormalizedgeom stats  (cost=0.00..332.26 rows=16326 width
            =52) (actual time=0.020..3.562 rows=16326 loops=1)
        ->  Index Scan using _I_points_dp on temp_geometrypoints points  (cost=0.14..75.22
            rows=1 width=50) (actual time=0.034..0.034 rows=0 loops=16326)
              Index Cond : (dp ~= stats.stat_geom)
              Filter : ((stats.id = id) AND (stats.an_id = an_id) AND st_equals(dp, stats.
                  stat_geom))
Planning Time: 0.566 ms
JIT:
  Functions: 13
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 2.193 ms, Inlining 28.026 ms, Optimization 254.567 ms, Emission 102.868
      ms, Total 387.654 ms
Execution Time: 976.350 ms
```

Listing 4.3: Query Plan for the optimized Join

4.0.1. Evaluation

The optimized query was implemented in the feedbase and measured. In figure 4.2 the duration of part 7 before and after the optimization is shown. In the optimized query, the duration dropped by a large amount and is at around 1s now. With this optimization, the very long durations of the database queries for the chart UI component could be eliminated. The target of the optimization is achieved for all chart queries, except one. Query 821 still has a duration of around two seconds.



(a) Reference

(b) Optimized Query

Figure 4.2.: Measurement of the duration of chart query parts before and after optimization displayed with a logarithmic scale.

5. Table UI Component Optimization

To analyse the table UI component in-depth the specific database query/queries that have a poor performance were identified first. There are five database queries in the table component. Three database queries are used for the table UI component in summary queries: SummaryNutrients, SummaryResults and SummaryCount. Two database queries are used in detail queries: DetailResults and DetailCount. DetailResults and SummaryResults are queries to retrieve actual rows of the table (in a paginated manner). DetailCount and SummaryCount are used to fetch the total count of rows for the specific query. Finally, SummaryNutrients is used to retrieve data about specific nutrients.

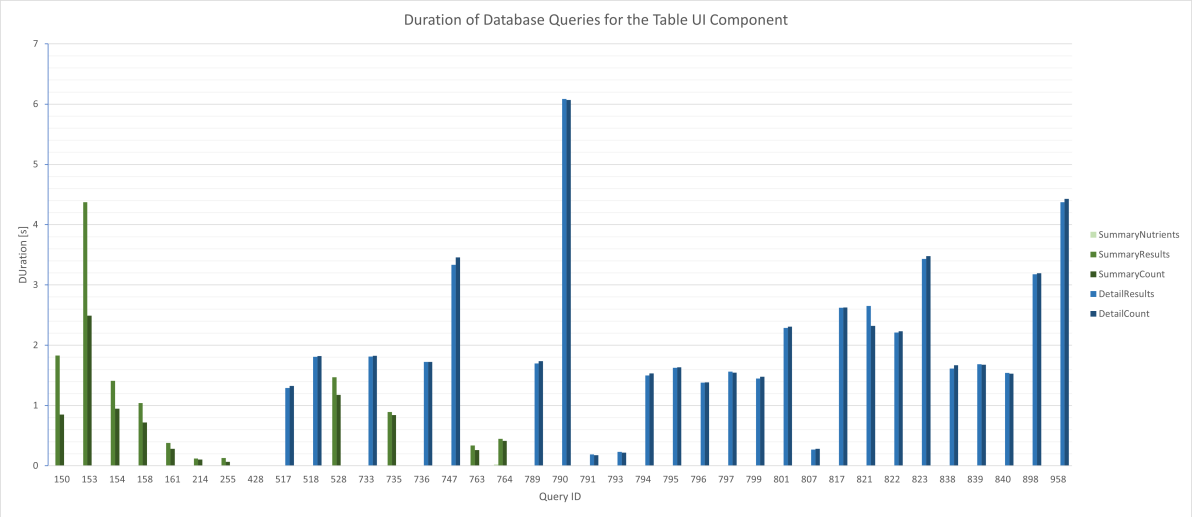


Figure 5.1.: Duration of database queries for the table UI component.

In figure 5.1 the duration of all database queries related to the table UI component is visualized. The measurements are taken from test-run 3. The queries with green bars are summary queries. The queries with blue bars are detail queries. SummaryNutrients is very fast. For summary queries, SummaryResults always takes longer than SummaryCount. For detail queries, DetailResults and DetailCount have similar durations. The underlying SQL statements of SummaryResults and SummaryCount are very similar. The first one returns the paginated rows while the second one only returns the count of all rows. The same can be observed for the detail database queries. Optimizing the summary/detail queries therefore can be done on either SQL statement while the performance improvement will be reflected on both of them.

5.1. Optimization for Detail Queries

For the analysis, a generated SQL statement for DetailResults in query 790 was used. This query took 6s to complete. The whole SQL statement is listed in the appendix in listing C.2. Executing the unchanged query on a PostgreSQL version 13 installation yielded a duration of around 750ms, opposed to 6s on PostgreSQL 9.6. In the query plans of the two PostgreSQL versions, there are some differences.

First, the newer PostgreSQL version uses parallel features, which are identified by the "Gather" node in the query plan excerpt in listing 5.1. The older PostgreSQL version uses no parallel features. To prove that this improves the performance, the parallel features were disabled manually and the execution times were measured. Disabling parallel features with `SET max_parallel_workers_per_gather = 0;` yielded durations twice as big as with parallel features enabled.

```
...
->  Hash Join  (cost=55554.42..106147.50 rows=1 width=86)
  Hash Cond: (fact_table_clean.id_sample_fkey = fact_table_clean_1.id_sample_fkey)
  ->  Gather  (cost=1000.00..51593.06 rows=6 width=90)
    Workers Planned: 2
    Workers Launched: 2
    ->  Parallel Seq Scan on fact_table_clean  (cost=0.00..50592.46 rows=2 width=90)
      Filter: ((id_nutrient_fkey = 112) AND (id_nutrient_analyses_fkey = 16))
      Rows Removed by Filter: 936024
  ->  Hash  (cost=54554.35..54554.35 rows=6 width=4)
    Buckets: 2048 (originally 1024)  Batches: 1 (originally 1)  Memory Usage: 64kB
    ->  Unique  (cost=54554.26..54554.29 rows=6 width=4)
      ->  Sort  (cost=54554.26..54554.27 rows=6 width=4)
        Sort Key: fact_table_clean_1.id_sample_fkey
        Sort Method: quicksort  Memory: 284kB
        ->  Gather  (cost=1000.57..54554.18 rows=6 width=4)
          Workers Planned: 2
          Workers Launched: 2
...
```

Listing 5.1: Excerpt from the Query Plan for DetailResults in PostgreSQL 13

Secondly, the newer version used JIT compilation. This can be seen based on the additional node "JIT" in the query plan summary that is highlighted in yellow in listing 5.2. When disabling JIT with `SET jit = off;` durations twice as big as with JIT compilation enabled were observed.

```
Planning Time: 1.597 ms
JIT:
  Functions: 68
  Options: Inlining false, Optimization false, Expressions true, Deforming true
  Timing: Generation 52.584 ms, Inlining 0.000 ms, Optimization 29.764 ms, Emission 171.603
      ms, Total 253.951 ms
Execution Time: 782.382 ms
```

Listing 5.2: Query Plan Summary for DetailResults in PostgreSQL 13

By disabling the new performance-enhancing features of PostgreSQL version 13, it was observed that each of the features helps to halve the duration of the selected query. Based on these findings the database was upgraded from version 9.6 to 13, being the newest available version at the time the thesis was written.

To assess the performance improvement from upgrading the PostgreSQL version a new test-run was executed with a PostgreSQL 13 installation. With the new test-run, it was assessed which database queries need optimization on the new version of PostgreSQL. The durations are visualized in figure 5.2. The overall picture looks pretty similar to the analysis of test-run 3 from above. The new features in PostgreSQL 13 helped to improve the performance of DetailCount and DetailResults. Query 790, which had a duration around 6s, dropped down to under a second on version 13. Some DetailCount and DetailResults queries still had durations that are similar to the durations before the update. Examples are queries 747 and 958. Further investigation therefore focused on query 747 for DetailCount and DetailResults.

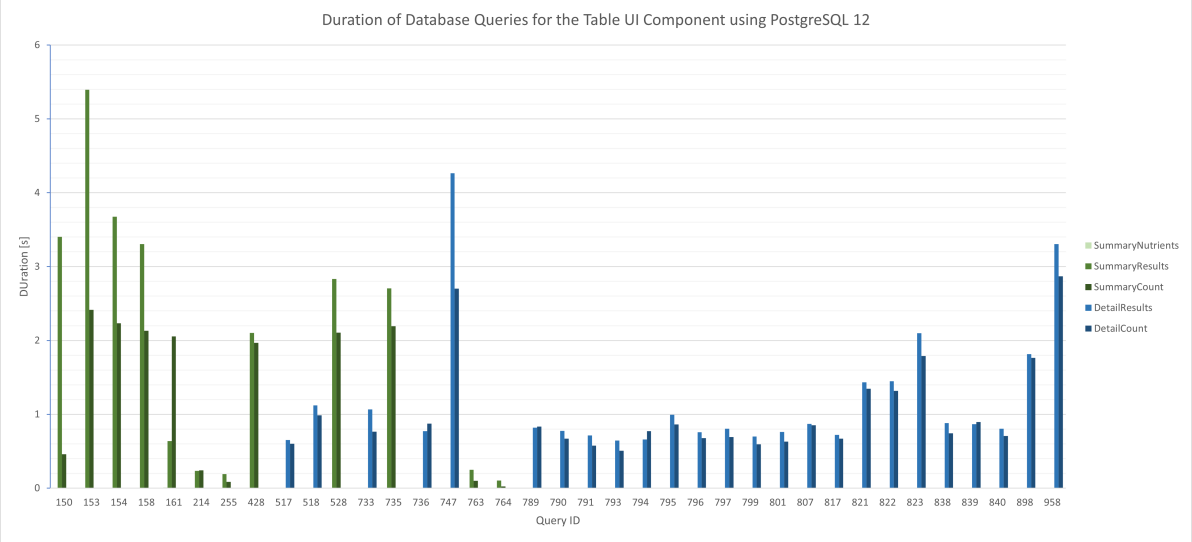


Figure 5.2.: Duration of database queries for the table UI component on PostgreSQL12.

In the query plan for DetailResults of query 747 row estimates, which are miles away from the effective row count, were found. An excerpt of the query plan is shown in listing 5.3. The part of the query plan corresponding to the join used for the `folded` CTE is shown. The listing is simplified and the relevant estimates, as well as actual row counts, are highlighted in yellow. The query planner did underestimate the count of the rows that are hashed (line 17). This underestimation comes from an underestimation of the rows returned by the CTE in line 20. The hashing thus needed more than 10 times the amount of buckets as originally planned (line 18). Additionally, the query planner overestimated the result rows of the hash join (line 8). This overestimation was propagated up to the sorting in line 4. Additionally, a lot of I/O is happening. The Hash Join needs to read 76399 + 2758 blocks from the shared cache (line 10, highlighted in orange).

```
1 GroupAggregate  (cost=903340.14..1013847.00 rows=1730400 width=256) (actual time
      =2469.735..2470.992 rows=50 loops=1)
2   Group Key: orderedbynutrient.row_number, fact_table_clean.lims_number, ...
3   Buffers: shared hit=76399, temp read=2758 written=3084
4   ->  Sort  (cost=903340.14..909665.05  rows=2529962 width=167) (actual time
        =2469.665..2470.280 rows=555 loops=1)
5     Sort Key: orderedbynutrient.row_number, fact_table_clean.lims_number, ...
6     Sort Method: external merge  Disk: 5576kB
```

```
7     Buffers: shared hit=76399, temp read=2758 written=3084
8     ->  Hash Left Join  (cost=199588.79..219198.03  rows=2529962 width=167) (actual time
          =1850.584..2358.126 rows=52208 loops=1)
9       Hash Cond: ((fact_table_clean.lims_number)::text = (orderedbynutrient.lims)::text)
10      Buffers:  shared hit=76399, temp read=2375 written=2385
11      ->  Finalize GroupAggregate  (cost=86809.78..105321.83 rows=86524 width=263) (actual
            time=1057.303..1538.151 rows=52208 loops=1)
12        Group Key: fact_table_clean.lims_number, fact_table_clean.id_nutrient_fkey,
              fact_table_clean.id_nutrient_analyses_fkey
13        Buffers: shared hit=9071, temp read=2126 written=2135
14        ->  Gather Merge  (cost=86809.78..98124.13 rows=81296 width=159) (actual time
              =1057.236..1356.435 rows=52208 loops=1)
15          Workers Planned: 2
16          Workers Launched: 2
17      ->  Hash  (cost=112705.91..112705.91 rows=5848 width=19) (actual time=793.260..793.410
            rows=20801 loops=1)
18        Buckets: 32768 (originally 8192)  Batches: 1 (originally 1)  Memory Usage: 1327kB
19        Buffers: shared hit=67328, temp read=249 written=250
20        ->  Subquery Scan on orderedbynutrient  (cost=112545.09..112705.91 rows=5848 width
              =19) (actual time=767.685..787.228 rows=20801 loops=1)
21          Buffers: shared hit=67328, temp read=249 written=250
```

Listing 5.3: Wrong Estimates in the Query Plan for DetailResults in PostgreSQL 12

To decrease the buffer usage and enhance the row estimations the query was rewritten to use temporary tables instead of CTEs. The resulting query ran in half the time (2s instead of 4.2s). An excerpt of the query plan is shown in listing 5.4. It is simplified and the relevant estimates, as well as actual row counts, are highlighted in yellow. The excerpt is showing the same join as above. The query plan shows significantly less buffer usage than in the query with CTEs. The join using CTEs read 76399 + 2758 blocks from the cache (listing 5.3, line 10), while the join using temporary tables read 133 + 640 blocks from the cache and 958 blocks from the disk storage (listing 5.4, line 10, highlighted in orange). The drastic decline of buffer usage is most likely responsible for the decrease of the duration. The wrong estimation is not much better compared to the query using CTEs.

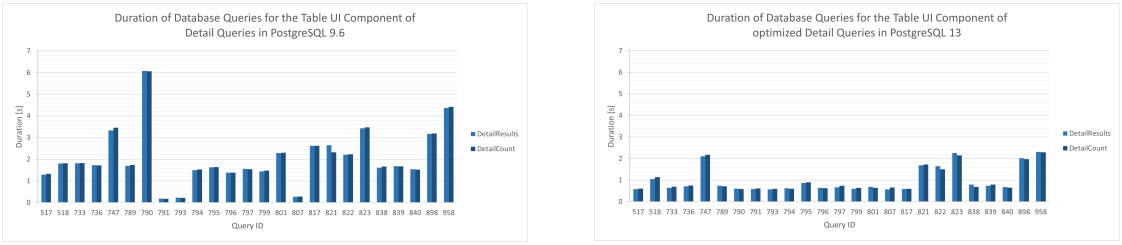
```
1   GroupAggregate  (cost=429291.32..477934.68 rows=459800 width=303) (actual time
        =312.014..387.534 rows=5000 loops=1)
2     Group Key: orderedbynutrient.row_number, filtered.lims_number, ...
3     Buffers: local hit=133 read=958 written=957, temp read=1337 written=1341
4     ->  Sort  (cost=429291.32..432578.60 rows=1314912 width=214) (actual time=311.952..329.564
          rows=52208 loops=1)
5       Sort Key: orderedbynutrient.row_number, filtered.lims_number, ...
6       Sort Method: external merge  Disk: 5576kB
7       Buffers: local hit=133 read=958 written=957, temp read=1337 written=1341
8       ->  Merge Right Join  (cost=6153.95..25992.30 rows=1314912 width=214) (actual time
            =153.915..219.369 rows=52208 loops=1)
9         Merge Cond: ((orderedbynutrient.lims)::text = (filtered.lims_number)::text)
10        Buffers:  local hit=133 read=958 written=957, temp read=640 written=642
11        ->  Sort  (cost=1018.39..1046.98 rows=11438 width=66) (actual time=80.057..83.177 rows
              =20801 loops=1)
12          Sort Key: orderedbynutrient.lims
13          Sort Method: quicksort  Memory: 2351kB
14          Buffers: local hit=133
15          ->  Seq Scan on orderedbynutrient  (cost=0.00..247.38  rows=11438 width=66) (actual
                time=0.007..3.083 rows=20801 loops=1)
16            Buffers: local hit=133
17        ->  Materialize  (cost=5135.56..5250.52 rows=22992 width=206) (actual time
              =49.826..82.596 rows=52208 loops=1)
```

```
18          Buffers: local read=958 written=957, temp read=640 written=642
19          ->  Sort  (cost=5135.56..5193.04 rows=22992 width=206) (actual time=49.821..69.883
                rows=52208 loops=1)
20            Sort Key: filtered.lims_number
21            Sort Method: external merge  Disk: 5120kB
22            Buffers: local read=958 written=957, temp read=640 written=642
23            ->  Seq Scan on filtered  (cost=0.00..1187.92 rows=22992 width=206) (actual time
                  =0.028..20.872 rows=52208 loops=1)
24              Buffers: local read=958 written=957
```

Listing 5.4: Excerpt of a Query Plan for DetailResults in PostgreSQL 12 using temporary tables instead of CTE's

5.1.1. Evaluation

In figure 5.3 the duration of the database queries of the table UI component for detail queries is shown. Only seven queries have durations of over 1 second. Three queries even take slightly longer than two seconds. Although the duration of the queries decreased significantly, the target of the optimization was not achieved for the table UI component of detail queries.



(a) Reference (b) Optimized Query

Figure 5.3.: Measurement of the duration of the table UI component for detail queries before and after the optimization.

5.2. Optimization for Summary Queries

For the optimization of summary queries, the SummaryResults SQL statement for query 153 was used. The full query can be found in listing C.3. The query uses 4 CTEs. To see how the execution time is distributed over the CTEs and the final select statement, the CTEs were substituted with temporary tables. In figure 5.4 the durations of all parts are shown. The focus of the optimization was on query parts 2 and 5 because they showed high durations.

5.2.1. Part 5

A closer look at figure 5.2 reveals that SummaryResults has significantly longer durations than SummaryCount, e.g. query 153. This difference in duration is interesting because the underlying SQL statements are very similar. SummaryResults returns the rows to be displayed,

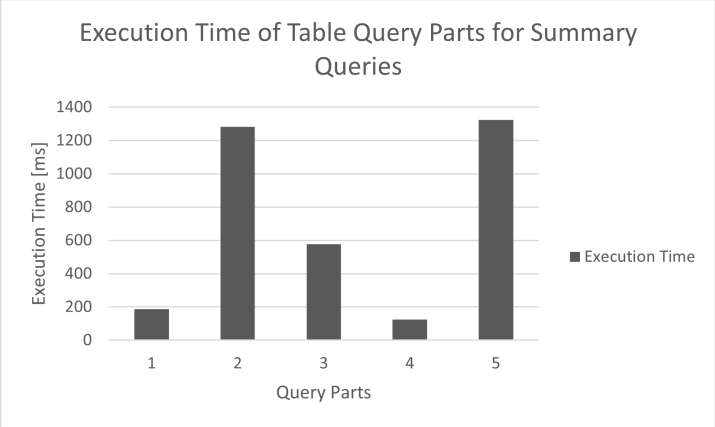


Figure 5.4.: Execution Time of Summary Query Parts

while SummaryCount returns the count of all rows. Based on figure 5.2 the SQL statements of query 153 is used for further analysis because it has long durations. When executing the SQL statement of SummaryResults durations of around 6 to 8 seconds occurred. When running the same statement with `EXPLAIN ANALYZE` durations of around 3 to 4 seconds were encountered. The PostgreSQL documentation states that there can be a difference between the duration of the statement and `EXPLAIN ANALYZE` of the same statement. The difference can be caused by network transmission costs and I/O conversion costs. These costs are not included in the measurements of `EXPLAIN ANALYZE`, because no rows are delivered to the client [4]. Based on this, the output produced by SummaryResults was analysed in detail.

```
1  SELECT feed_key, fname, json_agg(nutrient) AS nutrients
2  FROM (
3    SELECT feedkey AS feed_key, nid AS nutrient_key, coalesce(f.nutrient, r.nutrient) AS
         nutrient
4      FROM formulasfolded AS f FULL OUTER JOIN rowsfolded AS r
5       ON f.feed_key = r.feedkey AND f.nutrient_key = r.nid
6    ) as allNutrients
7    JOIN rows r ON r.feedkey = feed_key
8  GROUP BY feed_key, fname
9  ORDER BY fname DESC
10 LIMIT 50 OFFSET 0;
```

Listing 5.5: SQL statement for SummaryResults

In listing 5.5 the SQL statement of SummaryResults is shown. The `WITH` clause is excluded because the focus lies on the final output. The `SELECT` clause in line 1 shows that the output consists of three columns: `feed_key | fname | nutrients`. The `LIMIT` clause on the last line defines that a maximum of 50 rows is returned. This means that the rows need to be large to create high network transmission costs generating a noticeable difference in duration. Looking at the columns of the output rows the third column can be identified as a possible large column (in terms of size) because it is a json-object. The first and second columns cannot be very large, because they have a fixed upper memory limit. The data in the first column is an integer, while the data in the second is a varchar(255) string. The function `json_agg(nutrient)` creates a json array of nutrients for the feed. A closer look at the aggregated nutrients shows

that each nutrient is in the array multiple times. The reason for this duplicate data is found in the `FROM` clause in lines 2-7. First of all the CTE `formulasfolded` gets joined with the CTE `rowsfolded` over the feed and the nutrient key (highlighted in yellow). `formulasfolded` consists of all calculated nutrients for the requested feed. `rowsfolded` consists of all nutrients for the feed that are raw values. The subselect outputs all nutrients of a feed, each nutrient on a separate row. The column `nutrient` either consists of the raw value of the nutrient or the formula to calculate the nutrient. This is done by the `coalesce` function (highlighted in orange), which returns the first input argument that is not null. The input arguments `f.nutrient` and `r.nutrient` are either null or json objects. The results of the subselect get joined with `rows` (highlighted in blue). To better understand what goes on in this join an example is illustrated in table 5.1.

allNutrients

feed_key	nutrient_key	nutrient
900	3	nutrient_object_3
900	2	nutrient_object_2
900	5	nutrient_object_5
901	2	nutrient_object_2
901	10	nutrient_object_10

rows

nid	raw_value	feedkey	fname
3	101.3	900	Feed 900
2	null	900	Feed 900
5	22.014	900	Feed 900
2	1.2	901	Feed 901
10	null	901	Feed 901

allNutrients $\bowtie_{feed\_key=feedkey}$ **rows**

feed_key	nutrient_key	nutrient	nid	raw_value	feedkey	fname
900	3	nutrient_object_3	3	101.3	900	Feed 900
900	3	nutrient_object_3	2	null	900	Feed 900
900	3	nutrient_object_3	5	22.014	900	Feed 900
900	2	nutrient_object_2	3	101.3	900	Feed 900
900	2	nutrient_object_2	2	null	900	Feed 900
900	2	nutrient_object_2	5	22.014	900	Feed 900
900	5	nutrient_object_5	3	101.3	900	Feed 900
900	5	nutrient_object_5	2	null	900	Feed 900
900	5	nutrient_object_5	5	22.014	900	Feed 900
901	2	nutrient_object_2	2	1.2	901	Feed 901
901	2	nutrient_object_2	10	null	901	Feed 901
901	10	nutrient_object_10	2	1.2	901	Feed 901
901	10	nutrient_object_10	10	null	901	Feed 901

Table 5.1.: Example for the problematic join in the SQL statement for SummaryResults

`nutrient_object_x` is a json object with information about the nutrient with id x. `allNutrients` is the result of the subquery and consists of all nutrients for a feed with further information on the nutrient. `rows` consists of all nutrients for a feed with further information on the feed. To have information about the nutrients and the feed a join of the two relations is needed. The join is done with the join condition `feed_key = feedkey`. Thus the result consists of a row for each nutrient in `allNutrients` combined with every nutrient of the same feed from `rows`. The join result gets grouped by the columns `feedkey` and `fname`. The rows that are grouped are

shown with a yellow and orange background colour in the join result. All `nutrient_object_x` get aggregated into a json object using `json_agg()`. Thus with the current join condition, each `nutrient_object_x` is present in the output 3 times instead of once. For a feed with n nutrients, the output is an array that contains n^2 json objects instead of n.

For the visualization, each nutrient in `allNutrients` must be combined with the same nutrient of the same feed from `rows`. This would assure that there are no duplicate nutrients for a feed in the final result. To achieve that, the join condition can be extended with: `AND r.nid = nutrient_key`. The result of the new join condition is shown in table 5.2. With the new join condition the nutrient json aggregate consists of n objects instead of n^2.

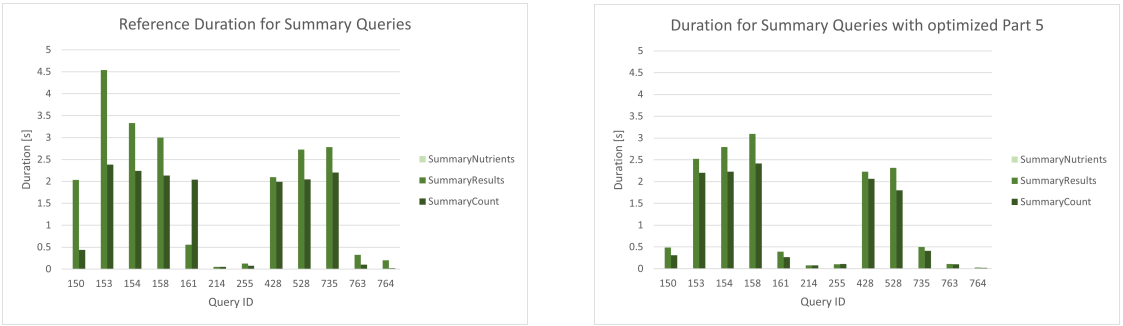
allNutrients $\bowtie_{feed\_key=feedkey \wedge r.nid=nutrient\_key}$ **rows**

feed_key	nutrient_key	nutrient	nid	raw_value	feedkey	fname
900	3	nutrient_object_3	3	101.3	900	Feed 900
900	2	nutrient_object_2	2	null	900	Feed 900
900	5	nutrient_object_5	5	22.014	900	Feed 900
901	2	nutrient_object_2	2	1.2	901	Feed 901
901	10	nutrient_object_10	10	null	901	Feed 901

Table 5.2.: Result of the optimized join in the SQL statement for SummaryResults

Evaluation

In figure 5.5 the measurements before and after the optimization of part 5 are shown. The difference of the duration of SummaryResults and SummaryCount has decreased significantly for many queries, e.g. query 153 shows a decrease of around 2s for SummaryCount. This is due to the smaller output in part 5 of the database query. The optimization of part 5 shows how important restrictive join conditions are. In this query, the join condition was formulated to open, which resulted in many duplicates. These duplicates get aggregated into a json-object. The size of the json-object showed a quadratic growth before the optimization. With the optimization, it was ensured that the json-object has a linear growing pattern.



(a) Reference (b) Optimized Part 5

Figure 5.5.: Measurement of the duration of the table UI component for summary queries before and after the optimization of part 5.

5.2.2. Part 2

In the query plan of part 2, a scan on the relation `d_nutrient` was identified to take 2/3 of the total duration. In listing 5.6 the corresponding part of the query plan is shown. The estimated costs are highlighted in yellow and are very low. Highlighted in orange the actual time is very large.

```
1  ->  Seq Scan on d_nutrient  (cost=0.00..17.74 rows=370 width=8) (
       actual time=806.132..806.368 rows=370 loops=1)
2    Filter: ((specie_id IS NOT NULL) AND (group_id IS NOT NULL))
3    Rows Removed by Filter: 4
```

Listing 5.6: Excerpt of a Query Plan for SummaryResults

Listing 5.7 shows the subquery in part 2 that executes the long scan. In this subquery three relations are joined. From the query plan, it is clear, that scanning relation `d_nutrient` is responsible for the long duration. In the join condition, there is one condition that checks if `d_nutrient.nutrient_key` is in a list of ids. The list of ids is constructed from strings, that contain comma-separated ids. The string function `regexp_split_to_table()` is used for that. In listing 5.7 the usages of `regexp_split_to_table()` are highlighted. To split a string by a single character PostgreSQL offers `unnest(string_to_array())`. In the query above, the result is the same as with `regexp_split_to_table()`. `regexp_split_to_table()` comes in handy if one wants to split a string by a sequence of specific characters. However, this additional functionality comes with a big trade-off in performance. Splitting by a regular expression has much higher costs than splitting by a sequence of characters. Hsu and Obe state that with increasing string-length the performance of `regexp_split_to_table()` gets worse, compared to the performance of `unnest(string_to_array())` [13].

```
30 SELECT
31   d_nutrient.nutrient_key AS nid,
32   reference_data.raw_value AS raw_value,
33   d_feed.feed_key AS feedkey,
34   COALESCE(d_feed.name_de, d_feed.name_en) AS fname
35 FROM
36   reference_data,
37   d_nutrient,
38   d_feed
39 WHERE
40   d_nutrient.nutrient_key = reference_data.nutrient_fkey
41   AND d_feed.feed_key = reference_data.feed_fkey
42   AND d_nutrient.specie_id IS NOT NULL
43   AND d_nutrient.group_id IS NOT NULL
44   AND d_feed.source = 'agroscope classified'
45   AND d_nutrient.nutrient_key IN (
46     SELECT DISTINCT id::integer FROM (
47       SELECT regexp_split_to_table(involved_nutrients_ids, E',') AS id FROM formulas
48       ) AS involved_ids WHERE id <> 'NULL' AND id IS NOT NULL
49     UNION
50       SELECT DISTINCT id::integer FROM (
51         SELECT regexp_split_to_table('251,195,...', E',') AS id FROM formulas
52     ) AS involved_ids WHERE id <> 'NULL' AND id IS NOT NULL
53   )
54   AND d_feed.feed_key IN (802,734,...)
55   AND (
56     reference_data.u_group_id = 1
57     OR reference_data.u_group_id = 0
58   )
```

```
59 ORDER BY feedkey, group_id, nid
```

Listing 5.7: Subquery of part 2 where a scan with a long duration is performed

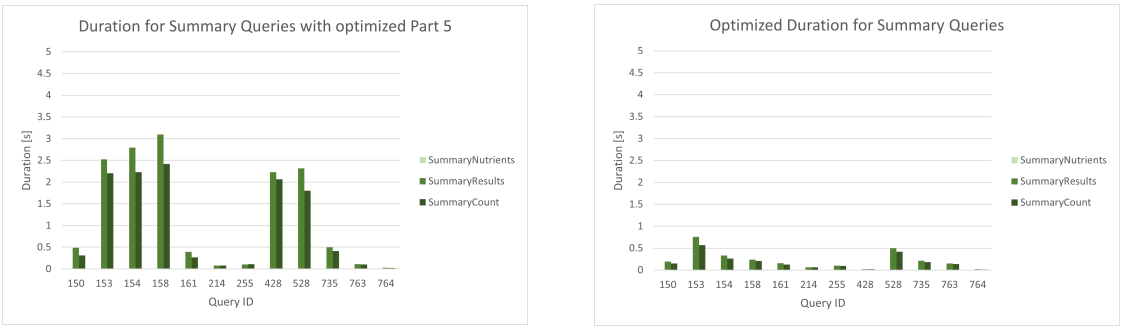
The usages of `regexp_split_to_table()` were rewritten to use `unnest(string_to_array())`. The resulting SQL command is shown in listing 5.8. The changes are highlighted in yellow. In query part 3 another occurrence of `regexp_split_to_table()` was found and replaced.

```
1 SELECT DISTINCT id::integer FROM (
2   SELECT  unnest(string_to_array(involved_nutrients_ids, E',')) AS id FROM formulas
3   ) AS involved_ids WHERE id <> 'NULL' AND id IS NOT NULL
4 UNION
5   SELECT DISTINCT id::integer FROM (
6   SELECT  unnest(string_to_array('251,195,...', E',')) AS id FROM formulas
7   ) AS involved_ids WHERE id <> 'NULL' AND id IS NOT NULL
```

Listing 5.8: Optimized creation of a list of ids from a string of ids

Evaluation

Figure 5.6 shows the duration of the database queries for the table UI component of summary queries before and after the optimization of part 2. The decrease in duration is very noticeable. Not a single query has a duration that is longer than a second. In fact, most of the queries have durations that are smaller than 0.5 seconds. The target of the optimization was achieved for the table UI component of summary queries. The optimization of part 2 showed how big the impact of expression evaluation on the performance can be. It is advisable to check computationally expensive expressions like `regexp_split_to_table()` for their performance with increasing input size. Bottle-necks can be eliminated by substituting the expression with a simpler one, that scales better (if there is a possible substitute).



(a) Reference

(b) Optimized Query

Figure 5.6.: Measurement of the duration of the table UI component for summary queries before and after the optimization of part 2.

6. Evaluation of the optimization

6.1. Overall Performance

Figure 6.1 shows the difference in duration of the system components for all queries before and after the optimization. The durations of D decreased a lot. Especially queries with long durations before the optimization, e.g. query 821 or 153, profit a lot from the optimization. The duration of timestamp D decreased by 8s in query 821 and by around 3.5 seconds in query 153. Summary queries show a decrease in AD, e.g. query 153. Because unnecessary data was removed from the output, the output of the database got smaller. This affects AD directly. C, CA and A show some minor differences. These differences are likely to be caused by natural fluctuations.

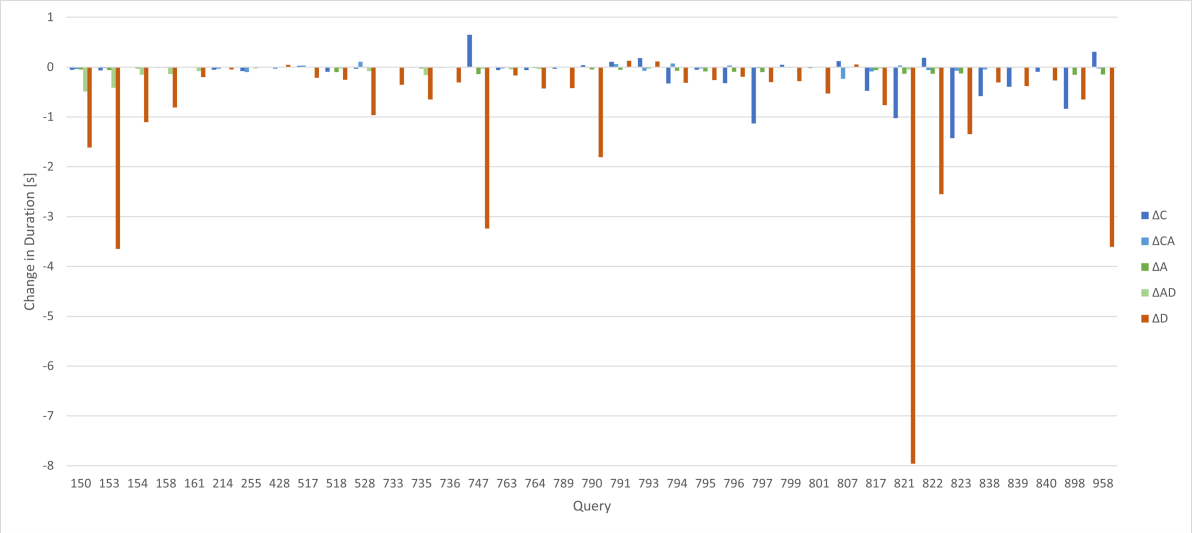


Figure 6.1.: Difference between durations of system components before and after the optimization.

The durations of D decreased a lot. In all the queries with long durations before the optimization a bottle-neck was found and eliminated. However the target of optimization was not achieved. The duration of seven table queries exceeds the threshold of one second. Furthermore, two map and one chart query have durations greater than a second. During the optimization, it became clear that one second is a target that is hard to achieve. Most of the queries need to join a handful of tables, with some tables that have a lot of data, e.g. 2 million tuples in `fact_table_clean`. The joining of these tables is already optimized by having indexes

in the right places. Here the question arises if it even is possible to optimize the large joins further and if yes, how big the gain in performance would be.

The mean of the measurements for the different timestamps is shown in figure 6.2. The mean of D decreased as expected. The same goes for the mean of AD. The changes in A, CA, and C are most likely caused by natural fluctuations and not a result of the optimization. The difference between the means of D and C after the optimization is much smaller than before the optimization. Because they are in the same order of magnitude now, further optimization must focus not only on D but also on C.

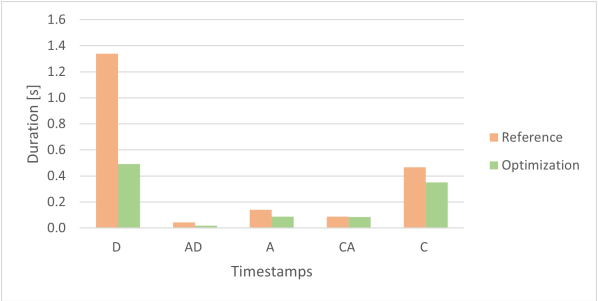
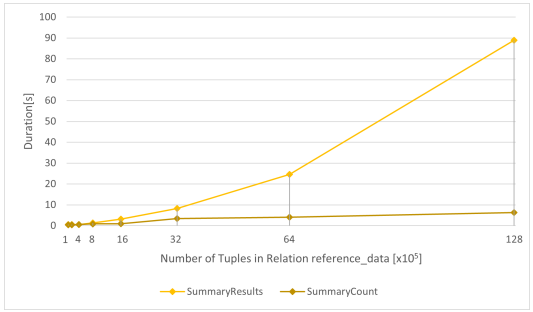


Figure 6.2.: Mean of the duration of all timestamps before and after the optimization.

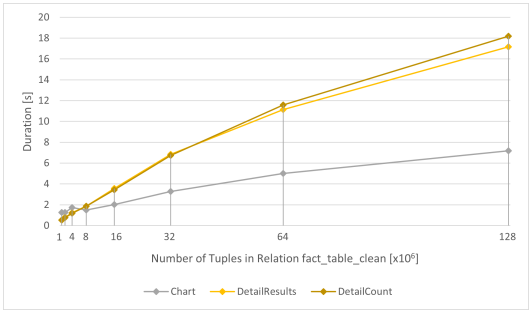
6.2. Scalability

The scalability of the optimized database queries was assessed. The goal of this evaluation was to see how the optimized queries handle larger input data. This is crucial because the data in the feedbase is growing steadily. To assess the scalability, tuples in two specific tables were duplicated. The result was a set of tables with a specific amount of tuples, e.g. tables with 1, 2, 4, 8, ... millions of tuples. For testing the scalability of the optimized queries, the tables `reference_data` and `fact_table_clean` were chosen. The tables were selected because they are used as inputs in the optimized queries. The measurements include I/O and network transmission time of the result rows. At the time of the scalability test, `reference_data` had around 250'00 tuples, while `fact_table_clean` had around 2.8 million tuples.

In figure 6.3a the durations of SummaryResults and SummaryCount with different sizes of the table `reference_data` are shown. SummaryResults grows faster than SummaryCount, which grows very slowly. The two underlying SQL statements of SummaryResults and SummaryCount are very similar, except for the returned data: SummaryResults returns the count of all rows, while SummaryCount returns a distinct amount of rows, i.e. 50 rows in the feedbase. The growth of SummaryResults is due to a growing result size, causing network transmission to take a long time. Even though SummaryResults always returns 50 rows, the rows themselves grow (in fact a json-object in one column of the result grows, as seen in section 5.2.1). Figure 6.3b shows the durations of DetailResults, DetailCount, and the chart database query with growing sizes of `fact_table_clean`. The chart database query grows slower than the table database queries. DetailResults and DetailCount grow similarly. Overall the scalability test revealed that the growth of the durations shows no signs of a strong growth, which would be an



(a) Summary Queries



(b) Detail Queries

Figure 6.3.: Results of a scalability test for database queries before and after optimization.

obvious clue for bad scalability. The test also showed that if the feedbase has 16 million tuples in `fact_table_clean` the chart query has a duration of 2 seconds, while the table queries take around 4 seconds. This means that with around six times more data the optimized database queries are still faster than the queries before the optimization with the original dataset.

7. Conclusion

This thesis evaluated the performance of the feedbase using an architecture-dependent understanding. A method of measuring performance in a multi-tier application was designed and implemented. The PostgreSQL database was identified as the main limiting part, approving the hypothesis of the evaluation. Additionally, the client tier was identified to be responsible for bad performance in rare cases. The approach to measure performance in a multi-tier application is applicable to any multi-tier application. It can be used as a guideline to implement performance measurements.

Using the findings of the performance evaluation the optimization focused on the PostgreSQL database. The detailed analysis of query plans was used to increase the performance of the database. In the process of optimization, the PostgreSQL database was updated from version 9.6 to version 13. The new version enhanced the performance because of the new performance-enhancing features included in the new version. The use of spatial indexes helped to speed up the database query for the chart UI component. By substituting CTEs with temporary tables the table UI component was optimized. Furthermore replacing some string functions improved performance. It was shown that the impact of expression evaluation on the performance of queries can be very big. Last but not least unnecessary output data was identified and removed, yielding a speed-up because of less I/O conversion costs.

Overall the optimization showed that is very important to keep the performance in mind when writing database queries. Small details can create bottle-necks, e.g. choosing CTE's instead of temporary tables or choosing `regexp_split_to_table()` instead of `unnest(string_to_array())`. Additionally, maintaining the system and updating all the components is of high importance to ensure that the most recent features can be used. However, it is always possible that a system has slow performance. In that case, it is important to use a systematic approach to analyse the system top-down to find the main limiting parts.

The performance of the database tier was enhanced a lot. In all queries with long durations, bottle-necks were identified and optimized. Nevertheless, there are still queries that don't reach the target threshold of one second. It was not assessed, whether these queries can be optimized further or if the threshold of one second is not realistic. The scalability tests showed that the optimized queries have different growing behaviours. No query showed a very fast growth. The results indicate that the database queries will still be performant with six times more data than now. However, the scalability tests are limited to specific database queries. Therefore a meaningful prediction on the scalability of the feedbase cannot be derived from the tests done in this thesis.

7.1. Future Work

Further investigation can be done on the target threshold of one second. It could be assessed if the threshold is reachable and if no, what a realistic threshold would look like. The thesis only assessed the performance of the initial visualization of the data in the feedbase. Future Work could explore the performance of interactions and optimize them if needed. Furthermore, the scalability of the feedbase could be assessed in total and not just for specific database queries, e.g. how much data the UI components can handle. The feedbase could be extended to guarantee good scalability. Because the performance of the optimized database tier is in the same order of magnitude as the performance of the client tier, it would be interesting to analyse the performance of the client tier in more detail. Finally, the performance measurements showed that the server tier has durations greater than a second for delivering the data for the map UI component. This specific part of the server tier could profit from optimization.

Bibliography

[1] The PostgreSQL Global Development Group. *PostgreSQL - Error Reporting and Logging*. Nov. 2020. URL: https://www.postgresql.org/docs/9.6/runtime-config-logging.html.

[2] Dominique Christina Hässig. "Development of Adaptive Heatmaps for Interactive Feed Explorations". BA thesis. University of Zurich, 2020.

[3] Mozilla Developer Network. *Date.prototype.toISOString() - JavaScript | MDN*. Nov. 2020. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date/toISOString.

[4] The PostgreSQL Global Development Group. *PostgreSQL - Using EXPLAIN*. Dec. 2020. URL: https://www.postgresql.org/docs/9.6/using-explain.html.

[5] The PostgreSQL Global Development Group. *PostgreSQL - What is JIT compilation?* Nov. 2020. URL: https://www.postgresql.org/docs/current/jit-reason.html.

[6] Rajsekhar Setaluri. *Why You Need Just In Time (JIT) Code Compilation*. Jan. 2021. URL: https://www.thoughtspot.com/codex/why-you-need-just-time-jit-code-compilation.

[7] Hans-Jürgen Schönig. *Mastering PostgreSQL 12*. Packt Publishing, 2019. ISBN: 1-83898-882-3.

[8] PostGIS Project. *PostGIS - Spatial Indexing*. Dec. 2020. URL: https://postgis.net/workshops/postgis-intro/indexing.html.

[9] Ralf Hartmut Güting. "An introduction to spatial database systems". In: *The VLDB Journal* 3.4 (Oct. 1994), pp. 357–399. ISSN: 0949-877X. DOI: 10.1007/BF01231602. URL: https://doi.org/10.1007/BF01231602.

[10] The PostgreSQL Global Development Group. *PostgreSQL - CREATE TABLE*. Dec. 2020. URL: https://www.postgresql.org/docs/current/sql-createtable.html.

[11] The PostgreSQL Global Development Group. *PostgreSQL - CREATE TABLE AS*. Dec. 2020. URL: https://www.postgresql.org/docs/13/sql-createtableas.html.

[12] Valentin Weiss. "Development of a Dynamic Web Application". BA thesis. University of Zurich, 2018.

[13] Leo Hsu and Regina Obe. *REGEXP_SPLIT_TO_TABLE AND STRING_TO_ARRAY UNNEST PERFORMANCE*. Jan. 2021. URL: http://www.postgresonline.com/journal/archives/370-regexp_split_to_table-and-string_to_array-unnest-performance.html.

Appendices

A. Glossary

A.1. Query Types

Query ID	Query Name
150	Energiereiche Einzelfutter für Schweine (FS)
153	Nährstoffprofil des Futterkataloges
154	Einzelfutter für Wiederkäuer
158	Grünfutter für Wiederkäuer
161	Essenzielle Aminosäuren
214	Futter reich an Omega-3 Fettsäuren
255	Weizen und Nebenprodukte
428	neue Fettsäurenanalytik_inArbeit
528	Essentielle Aminosäuren in Raufutter
735	verdauliche Aminosäuren Schwein in TS
763	Gerste und Nebenprodukte
764	Hafer und Nebenprodukte

Table A.1.: Summary Queries and their ID's

Query ID	Query Name
517	Luzernemehl Streuung im Rohproteingehalt
518	Luzerneheu Praxis
733	Kartoffelprotein RP-Gehalt
736	Rapskuchen: RP, RL
747	Grassilage Praxis
789	Maiskleber, RP-Gehalt
790	Proteinerbsen, RP und RF
791	Rapssaat standard, RL, RLGC, RP
793	Rapssaat HOLL, MUFA, PUFA
794	Triticale, VES-Gehalt
795	Gerste P-Gehalt rückläufig
796	Gerste Fett-Gehalt f(Methode)
797	Maiskörner Fettparameter nach Kanton
799	Futterrübe, Faserfraktionen
801	Sojakuchen, RL und RP
807	Sonnenblumenkerne HO
817	Korrelation RP und RLGC>2011 in Rapssamen standard und HOLL
821	regionaler NEL-Gehalt in Dürrfutter 2005-2017
822	Heu 1. Schnitt: Zucker regionale Verteilung und Korrelation zu ADF
823	Bergheu > 1000 m, 2005-2017
838	Rapskuchen, Korrelation RP und RL
839	DCP, Ca und P
840	Sonnenblumenschrot, Korrelation RP und RF
898	Heuernte 2018
958	Maissilage Praxis

Table A.2.: Detail Queries and their ID's

B. Performance Analysis

B.1. System Component Performance Test-run

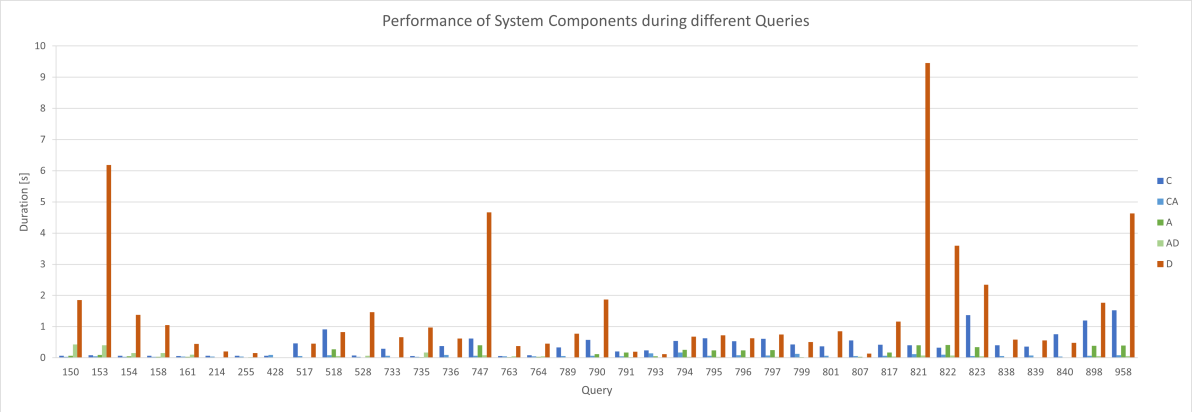


Figure B.1.: Test-run 1: Performance of system components during different queries

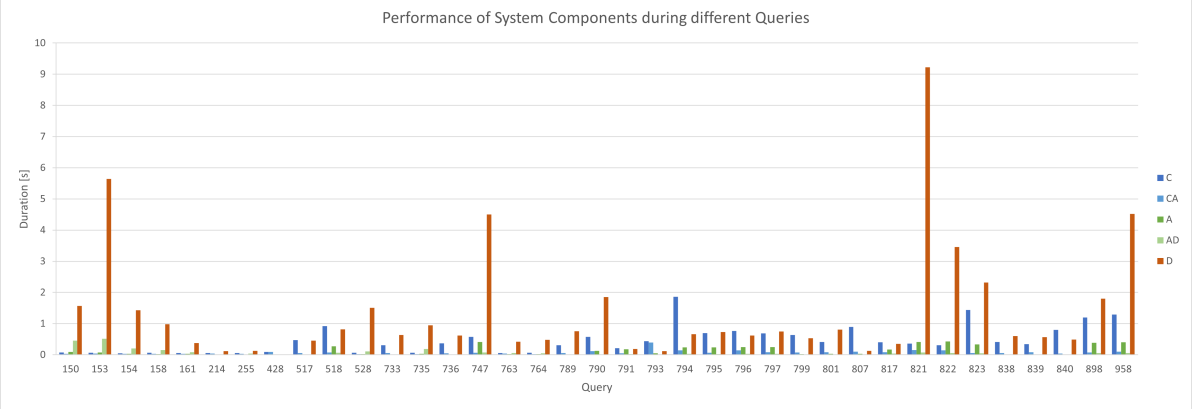


Figure B.2.: Test-run 2: Performance of system components during different queries

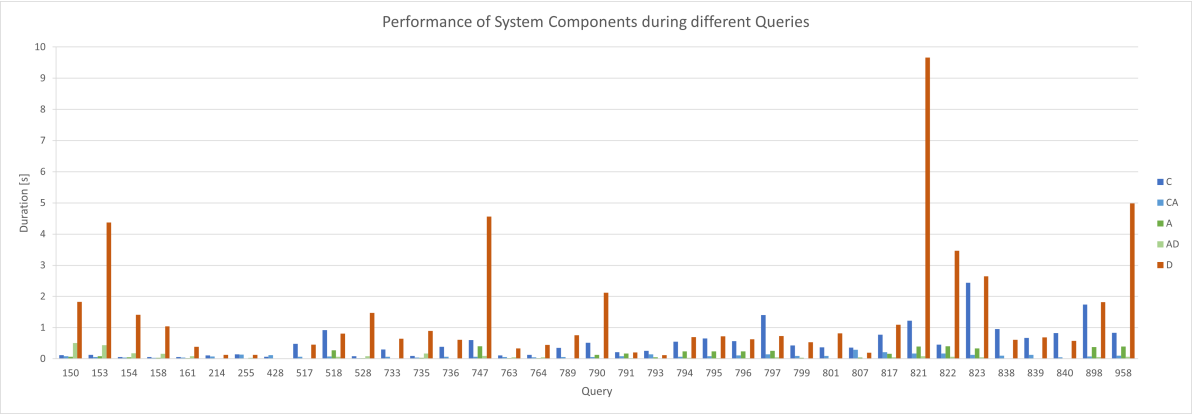


Figure B.3.: Test-run 3: Performance of system components during different queries

B.2. Statistical Analysis

Timestamp	Test-run 1: mean	Test-run 2: mean	Test-run 3: mean
D	1.380	1.340	1.418
AD	0.040	0.043	0.045
A	0.139	0.142	0.139
CA	0.075	0.089	0.102
C	0.408	0.469	0.526

Table B.1.: Statistical parameters for the differential duration of all three test-runs

C. SQL Statements

```sql
WITH
  nutrients AS (
  SELECT
    lims_number,
    id_nutrient_fkey AS id,
    id_nutrient_analyses_fkey AS an_id,
    AVG(quantity) AS avg_quantity,
    MAX(COALESCE(t_day, to_date(t_year||'-01-01', 'YYYY-MM-DD'))) AS day,
    MIN(season_en) season,
    MIN(COALESCE(d_feed.name_en, d_feed.name_de)) as feedname,
    MIN(feed_key) as feedkey
    FROM fact_table_clean
      JOIN d_feed ON id_feed_fkey = feed_key
      JOIN d_nutrient ON id_nutrient_fkey = nutrient_key
      JOIN d_origin ON id_origin_fkey = origin_key
      JOIN d_time ON id_time_fkey = time_key
      JOIN d_nutrient_analyses ON id_nutrient_analyses_fkey = nutrient_analyses_key
    WHERE
      id_sample_fkey in (SELECT * FROM (
        SELECT distinct id_sample_fkey
        FROM fact_table_clean
        JOIN d_time ON id_time_fkey = time_key
        JOIN d_origin ON id_origin_fkey = origin_key
        WHERE
        lims_number <> '0-const'
        AND (id_feed_fkey IN (3,12,2,1)) AND (id_nutrient_fkey IN ('133')) AND
            id_nutrient_analyses_fkey IN ('7') AND d_origin.canton IN ('Aargau','Appenzell
            Ausserrhoden','Appenzell Innerrhoden','Baselland','Bern','Fribourg','Glarus','
            Graubünden','Jura','Luzern','Neuchâtel','Nidwalden','Obwalden','Schaffhausen','
            Schwyz','Solothurn','St. Gallen','Thurgau','Ticino','Uri','Vaud','Wallis','Zug','
            Zürich','n/a') AND ((d_time.moment = 1 OR d_time.moment = 2) AND d_time.t_year IN
             ('2005','2006','2007','2008','2009','2010','2011','2012','2013','2014','2015','
            2016','2017'))
      ) AS all_lims)
      AND (id_feed_fkey IN (3,12,2,1)) AND (id_nutrient_fkey IN ('133')) AND
          id_nutrient_analyses_fkey IN ('7') AND d_origin.canton IN ('Aargau','Appenzell
          Ausserrhoden','Appenzell Innerrhoden','Baselland','Bern','Fribourg','Glarus','Graub
          ünden','Jura','Luzern','Neuchâtel','Nidwalden','Obwalden','Schaffhausen','Schwyz','
          Solothurn','St. Gallen','Thurgau','Ticino','Uri','Vaud','Wallis','Zug','Zürich','n/
          a') AND ((d_time.moment = 1 OR d_time.moment = 2) AND d_time.t_year IN ('2005','
          2006','2007','2008','2009','2010','2011','2012','2013','2014','2015','2016','2017')
          )
    GROUP BY lims_number, id_nutrient_fkey, id_nutrient_analyses_fkey
    ORDER BY lims_number
),
-- Get the avg_quantity, day, global MIN and MAX values
nutrientStats AS (
  SELECT
    id,
    an_id,
    avg_quantity,
    day,
    (SELECT MIN(day) FROM nutrients) AS min_day,
    (SELECT MAX(day) FROM nutrients) AS max_day,
    (SELECT MIN(avg_quantity) FROM nutrients) AS max_quantity,
    (SELECT MAX(avg_quantity) FROM nutrients) AS min_quantity
```

```
    FROM nutrients
),

-- normalize the avg_quantity and day dimensions
statsNormalized AS (
  SELECT
    id,
    an_id,
    avg_quantity,
    day,
    (EXTRACT(EPOCH FROM day - to_timestamp(0)) - EXTRACT(EPOCH FROM min_day - to_timestamp(0)
        )) / ((EXTRACT(EPOCH FROM max_day - to_timestamp(0)) - EXTRACT(EPOCH FROM min_day -
        to_timestamp(0))) + ((EXTRACT(EPOCH FROM max_day - to_timestamp(0)) - EXTRACT(EPOCH
        FROM min_day - to_timestamp(0)))=0)::integer) AS day_normalized,
    (avg_quantity - min_quantity) / ((max_quantity - min_quantity) + ((max_quantity -
        min_quantity)=0)::integer) AS quantity_normalized
    FROM nutrientStats
),

-- Create Proximity Clusters of these points and overlay them with a concave hull
clustered AS (
  SELECT
    id,
    an_id,
    ST_ConcaveHull(unnest(ST_ClusterWithin(ST_MakePoint(
    day_normalized,
    quantity_normalized
    ), 0.005)), 0.90, true) AS geometry
    FROM statsNormalized
    GROUP BY id, an_id
),

-- Enumerate the clusters
clusteredEnumerated AS (
    SELECT ROW_NUMBER() OVER (ORDER BY geometry) as geom_number, id, an_id, geometry FROM
        clustered
),

-- Split the cluster geometries to points
geometryPoints AS (
  SELECT
    geom_number,
    id,
    an_id,
    (ST_DumpPoints(geometry)).geom AS dp
    FROM clusteredEnumerated
),

 geometriesWithCenters AS (
  SELECT id, an_id,
    --ST_Centroid(points) AS center,
    -- case: linestring
    CASE WHEN COUNT(points) = 2 THEN
    ST_MakeLine(points)
    WHEN COUNT(points) > 2 THEN
    -- case: polygon
    ST_MakePolygon(ST_AddPoint(ST_MakeLine(points), ST_GeometryN(ST_Collect(points), 1)))
    ELSE
    -- case: point
    ST_Collect(points)
    END AS geometry
    FROM (
      SELECT geom_number,
      points.id AS id,
      points.an_id AS an_id,
      ST_MakePoint(EXTRACT(EPOCH FROM day - to_timestamp(0)) * 1000, avg_quantity) AS points
```

```
      FROM geometryPoints AS points, statsNormalized AS stats
      WHERE points.id = stats.id
        AND points.an_id = stats.an_id
        AND ST_Equals(dp, ST_MakePoint(day_normalized, quantity_normalized))
    ) as geoms
    GROUP BY id, an_id, geom_number
)

-- final json formatting for the client
SELECT
  id, an_id,
  json_agg(geometry) AS geometries
  FROM (
    SELECT
    id, an_id,
    json_build_object('geometry', geometry, 'center', center) AS geometry
    FROM (
      SELECT
      id, an_id,
      ST_AsGeoJson(geometry)::json AS geometry,
      ST_AsGeoJson(ST_Centroid(geometry))::json AS center
      FROM geometriesWithCenters
    ) as withCenters
  ) as json
  GROUP BY id, an_id
```

Listing C.1: Chart query with a duration of over 20s

```
WITH filtered AS (
    SELECT
    lims_number,
    id_nutrient_fkey AS id,
    id_nutrient_analyses_fkey AS an_id,
    MAX(priority) AS priority,
    CASE WHEN AVG(quantity) > 1 THEN
       rtrim(ROUND(AVG(quantity)::numeric, 3)::text, '0')::numeric
    ELSE rtrim(ROUND(AVG(quantity)::numeric, 5)::text, '0')::numeric
    END AS avg_quantity,
    MAX(COALESCE(t_day, to_date(t_year||'-01-01', 'YYYY-MM-DD'))) AS day,
    MIN(postal_code) AS postal_code,
    MIN(origin_key) AS origin_key,
    MIN(altitude_class) AS altitude,
    MIN(season_de) season,
    MIN(canton) AS canton,
    MIN(COALESCE(d_feed.name_de, d_feed.name_en)) as feedname,
    MIN(feed_key) as feedkey,
    MIN(latitude) as latitude,
    MIN(longitude) as longitude
    FROM fact_table_clean
    JOIN d_feed ON id_feed_fkey = feed_key
    JOIN d_nutrient ON id_nutrient_fkey = nutrient_key
    JOIN d_origin ON id_origin_fkey = origin_key
    JOIN d_time ON id_time_fkey = time_key
    JOIN d_nutrient_analyses ON id_nutrient_analyses_fkey = nutrient_analyses_key
    WHERE
    id_sample_fkey in (SELECT * FROM (
      SELECT distinct id_sample_fkey
      FROM fact_table_clean
        JOIN d_time ON id_time_fkey = time_key
        JOIN d_origin ON id_origin_fkey = origin_key
      WHERE
        lims_number <> '0-const'
        AND (id_feed_fkey IN (834)) AND (id_nutrient_fkey IN
            (112,180,144,158,163,160,159,142))
    ) AS all_lims)
      AND (id_feed_fkey IN (834)) AND (id_nutrient_fkey IN (112,180,144,158,163,160,159,142))
```

```
      GROUP BY lims_number, id_nutrient_fkey, id_nutrient_analyses_fkey
      ORDER BY lims_number
    ),

         orderedByNutrient AS (
         SELECT
         ROW_NUMBER() OVER (ORDER BY AVG(quantity) DESC),
         lims_number AS lims
         FROM fact_table_clean
         JOIN d_feed ON id_feed_fkey = feed_key
         JOIN d_nutrient ON id_nutrient_fkey = nutrient_key
         JOIN d_origin ON id_origin_fkey = origin_key
         JOIN d_time ON id_time_fkey = time_key
         JOIN d_nutrient_analyses ON id_nutrient_analyses_fkey = nutrient_analyses_key
         WHERE
         id_sample_fkey in (SELECT * FROM (
         SELECT distinct id_sample_fkey
         FROM fact_table_clean
           JOIN d_time ON id_time_fkey = time_key
           JOIN d_origin ON id_origin_fkey = origin_key
         WHERE
           lims_number <> '0-const'
           AND id_nutrient_fkey = 112
           AND id_nutrient_analyses_fkey = 16
         ) AS all_lims)
          AND id_nutrient_fkey = 112
          AND id_nutrient_analyses_fkey = 16
         GROUP BY lims_number
       ),

       folded AS (
         SELECT
          -- lims number only visible for admin users
         regexp_replace(lims_number, '\S+(\S$)', 'xxx-\1', 'g') AS lims_number,
         array_agg(id) AS ids,
         array_agg(an_id) AS an_ids,
         array_agg(avg_quantity) AS avg_quantities,
         feedname,
         feedkey,
         season,
         canton,
         postal_code,
         day,
          false as highlight
          FROM filtered
         LEFT JOIN orderedByNutrient ON lims_number = orderedByNutrient.lims

         GROUP BY lims_number, feedname, feedkey, season, canton, postal_code, day ,
             row_number
         ORDER BY row_number
       )
    SELECT * FROM folded LIMIT 50 OFFSET 0
```

Listing C.2: DetailResults query with a duration of around 6s

```
    WITH formulas AS (
      SELECT
          id_feed AS feed_key,
          nutrient_fkey AS nutrient_key,
          regexp_replace(expanded_formula_eval, 'coalesce\(.[^+*/()-]{1,30}','(','g' ) AS
              expanded_formula_eval,
          involved_nutrients_ids,
          correct
          FROM
            t_formula_feed
            JOIN t_formula ON t_formula_feed.id_formula = t_formula.id
```

```
            JOIN d_feed ON d_feed.feed_key = t_formula_feed.id_feed
        WHERE
            t_formula.nutrient_fkey IN
                (251,195,129,120,122,40,180,127,1,117,206,136,86,5,194,132,176,178,142,160,
                154,133,158,248,88,145,163,6,159,186,234,141) AND
            d_feed.feed_key IN
                (802,734,889,745,746,738,739,806,774,741,733,811,747,748,753,754,758,752,763,
                 789,762,769,772,778,773,749,750,759,760,780,775,807,856,801,810,768,814,791,
                 793,809,786,798,795,785,790,815,812,799,832,792,796,797,803,804,808,824,829,
                 835,842,843,846,822,847,816,826,825,841,823,800,830,831,836,837,838,839,852,
                 853,864,868,851,911,910,865,714,859,888,862,909,858,875,876,850,854,861,855,
                 884,885,886,887,894,906,900,899,848,881,891,883,902,873,901,863,867,702,712,
                 716,699,695,719,697,720,715,711,713,722,717,721,874,710,890,725,905,908,727,
                 840,784,898,897,845,687,686,691,805,1298,878,685,766,827,907,1321,893,705,
                833,895,698,735,751,849,742,740,903,880,813,828,877,708,709,776,781,857,860,
                782,767,736,770,783,761,821,871,696,688,684,692,707,703,706,700,701,704,726,
                723,724,779,689,693,694,732,728,730,731,690,744,755,756,764,771,777,765,834,
                844,879,892,896,904,817,818,819,820,729,743,737,869,870,882,718,787,788, 872,
                 912,757) AND
            t_formula.abbr_generic_de IS NOT NULL AND
            (rtrim(t_formula.abbr_generic_de) NOT LIKE '%TS]')
    ),

    rows AS (
        SELECT
        nid,
          CASE WHEN raw_value > 1 THEN
        rtrim(ROUND(raw_value::numeric, 3)::text, '0')::numeric
        ELSE
        rtrim(ROUND(raw_value::numeric, 5)::text, '0')::numeric
        END AS raw_value,
          feedkey,
          fname
        FROM (
          SELECT
        d_nutrient.nutrient_key AS nid,
          reference_data.raw_value AS raw_value,
          d_feed.feed_key AS feedkey,
          COALESCE(d_feed.name_de, d_feed.name_en) AS fname
        FROM
        reference_data,
          d_nutrient,
          d_feed
        WHERE
        d_nutrient.nutrient_key = reference_data.nutrient_fkey
        AND d_feed.feed_key = reference_data.feed_fkey
        AND d_nutrient.specie_id IS NOT NULL
        AND d_nutrient.group_id IS NOT NULL
        AND d_feed.source = 'agroscope classified'
        AND d_nutrient.nutrient_key IN (
        SELECT DISTINCT id::integer FROM (
            SELECT regexp_split_to_table(involved_nutrients_ids, E',') AS id FROM
                formulas
            ) AS involved_ids WHERE id <> 'NULL' AND id IS NOT NULL
        UNION
        SELECT DISTINCT id::integer FROM (
            SELECT regexp_split_to_table('
                251,195,129,120,122,40,180,127,1,117,206,136,86,5,194,132,176,178,142,160,
                 154,133,158,248,88,145,163,6,159,186,234,141', E',') AS id FROM formulas
            ) AS involved_ids WHERE id <> 'NULL' AND id IS NOT NULL
        )
          AND d_feed.feed_key IN
                (802,734,889,745,746,738,739,806,774,741,733,811,747,748,753,754,758,752,763,
                 789,762,769,772,778,773,749,750,759,760,780,775,807,856,801,810,768,814,791,
                 793,809,786,798,795,785,790,815,812,799,832,792,796,797,803,804,808,824,829,
                 835,842,843,846,822,847,816,826,825,841,823,800,830,831,836,837,838,839,852,
```

```
                853,864,868,851,911,910,865,714,859,888,862,909,858,875,876,850,854,861,855,
                884,885,886,887,894,906,900,899,848,881,891,883,902,873,901,863,867,702,712,
                716,699,695,719,697,720,715,711,713,722,717,721,874,710,890,725,905,908,727,
                840,784,898,897,845,687,686,691,805,1298,878,685,766,827,907,1321,893,705,
               833,895,698,735,751,849,742,740,903,880,813,828,877,708,709,776,781,857,860,
               782,767,736,770,783,761,821,871,696,688,684,692,707,703,706,700,701,704,726,
               723,724,779,689,693,694,732,728,730,731,690,744,755,756,764,771,777,765,834,
               844,879,892,896,904,817,818,819,820,729,743,737,869,870,882,718,787,788,872,
               912,757)
          AND (
            reference_data.u_group_id = 1
          OR reference_data.u_group_id = 0
        )
          ORDER BY feedkey, group_id, nid
        ) aa
        ),
formulasfolded AS (
   SELECT feed_key, nutrient_key,
   json_build_object(
             'id', nutrient_key,
     'involved_nutrients', json_agg(json_build_object(
             'nutrient_id',involved_id,
             'raw_value', raw_value
     )),
     'formula', expanded_formula_eval) as nutrient
     FROM (
      SELECT feed_key, nutrient_key, involved_id::integer, expanded_formula_eval
      FROM (
            SELECT feed_key, nutrient_key,
           regexp_split_to_table(involved_nutrients_ids, E',') AS involved_id,
           expanded_formula_eval
           FROM formulas
     ) AS split
      WHERE involved_id <> 'NULL' AND involved_id <> ''
   ) AS mapping
   JOIN rows ON feedkey = feed_key AND nid = involved_id
   GROUP BY feed_key, nutrient_key, expanded_formula_eval

   ),
   rowsfolded AS (
   SELECT
   feedkey,
   nid,
   json_build_object(
             'id', nid,
     'involved_nutrients', json_agg(json_build_object(
             'nutrient_id',nid,
             'raw_value', raw_value
     )),
     'formula', NULL) as nutrient
     FROM rows
   GROUP BY feedkey,nid,fname)



   SELECT feed_key, fname, json_agg(nutrient) AS nutrients
   FROM (
     SELECT feedkey AS feed_key, nid AS nutrient_key, coalesce(f.nutrient, r.nutrient) AS
         nutrient
         FROM formulasfolded AS f FULL OUTER JOIN rowsfolded AS r
         ON f.feed_key = r.feedkey AND f.nutrient_key = r.nid
   ) as allNutrients
   JOIN rows r ON r.feedkey = feed_key
```

```
111         GROUP BY feed_key, fname
112         ORDER BY fname DESC
113         LIMIT 50 OFFSET 0;
```

Listing C.3: SummaryResults query