Development of a Dynamic Web Application

The Swiss Feed Database

Bachelor's Thesis in Informatics

submitted by
Valentin Weiss
Student ID Number 15-707-870

Completed at the
Department of Informatics
of the University of Zurich
Prof. Dr. M. Böhlen

Supervisors: Georgios Garmpis & Prof. Dr. M. Böhlen
Submission Date: 11.06.2018

Acknowledgments

I hereby would like to thank Prof. Dr. Michael Böhlen for his excellent guidance through the entire work and development process. His inputs, explanations to complex problems and useful feedback helped me tremendously in realizing my ideas and the specific requirements of the application.
I would also like to thank Georgios Garmpis for introducing me to this project and developing and discussing ideas on how to address the challenges of this application.
Another special thanks to Annelies Bracher for her useful feedback and input from the user's perspective.

Abstract

This thesis explores data summarization techniques for the feed data of the Swiss Feed Database. It proposes and evaluates solutions to reduce the client-side load by reducing the amount of tuples passed to the client. Most of the work is shifted away from the client to the database layer following the thin-client paradigm with the aim of creating a stable and scalable solution. The summarization techniques were optimized to best suit the data visualization at hand. This includes techniques such as providing only a partial view of the data, aggregating the data set or leveraging properties of the spatial distribution of data points to abstract groups of data points into geometric shapes.

Zusammenfassung

Diese Arbeit untersucht Datenzusammenfassungstechniken für die Futterdaten der Schweizer Futtermitteldatenbank. Es werden Lösungen zur Reduzierung der Client-seitigen Belastung vorgeschlagen und bewertet indem die Anzahl Tupel, die der Client verarbeiten muss, reduziert wird. Dem Thin-Client-Prinzip folgend, wird der grösste Teil der Arbeit vom Client auf die Datenbankebene verlagert mit dem Ziel, eine stabile und skalierbare Lösung zu schaffen. Die Zusammenfassungstechniken wurden für die jeweilige Datenvisualisierung optimiert. Dazu gehören Techniken wie die Bereitstellung einer Teilansicht der Daten, die Aggregation des Datensatzes oder die Nutzung der Eigenschaften der räumlichen Verteilung von Datenpunkten um Gruppen von Datenpunkten zu geometrischen Formen zu abstrahieren.

Contents

1 Introduction

The Swiss Feed Database (TDFB) is a web application where nutrient measurements of different feeds are stored. The stored data can be filtered and visualized using different visualizations such as a table, different types of charts and a map.
The underlying database is continuously updated with new measurements by Agroscope, the Swiss Center for Agricultural Research.

1.1 Challenges

The database is updated regularly with new measurements resulting in ever-growing relations. The current application executes one query for every request made by the client and returns the result set to the client. The result set is then used to fill a table, a map and several charts with data. There are certain queries which return well over 10'000 tuples and can not be handled by a browser anymore. These queries cause a browser hang or even cause the browser to crash. This case is currently handled by retrieving a random selection and put a limit on the amount of tuples that can be retrieved from the database.
This approach comes with limitations. As the amount of tuples matching a given query grows, the limited result set is not guaranteed to be representative for the whole result set. In fact it will become more unlikely to be representative with an ever-growing amount of measurement data. Furthermore there is no way to guarantee that a given limit (e.g., maximal 2000 tuples) on a result set will not cause a client to hang or even crash. This work addresses this issue by reducing the amount of tuples retrieved from the database using dedicated queries for each visualization with the aim of reducing the tuple count passed to the client. Clients can be desktop or mobile devices with different specifications and processing power. A desktop machine might be able to handle a given result set while a mobile device might crash. Scalability is thus an issue which is not well addressed in the current implementation.

1.1.1 Requesting Data

In order to get a better understanding of performance in terms of time when requesting data, a few test queries to retrieve nutrient measurements were run in a local development environment with a basic web server setup to return result tuples in JSON format. The tuples were used by the client to render an HTML table where each tuple represents a row.

t_{db}	t_{server}	t_{client}	n_{tuples}
1.21s	1.64s	12.1s	5000
4.1s	2.39s	no response	10000
5.6s	12.23s	no response	20000
6.5s	58.48s	no response	30000
7.2s	no response	no response	40000

Table 1: Database and server response times

The database time (t_{db}) was measured using pgAdmin 3 while the server time (t_{server}) denotes the time needed to return a response by the local web server. The client (browser) stops responding between 5'000 - 10'000 tuples. At around 30'000 - 40'000 tuples the load time of the server becomes indefinitely long. An interesting observation comes from the comparison of database and server time. The database has no problems fetching up to 40'000 tuples. The web server, however, shows a huge performance drop from 20'000 to 30'000 returned tuples.

1.2 Problem Definition

In the current version the user selects filtering criteria and the results are fetched from the server in one HTTP request/response round trip. The result set returned to the client is then used to render all visualizations at the same time. The benefit of this approach is that the server is only queried once per user-defined query search. The downside of this approach, however, shows when a query to the server returns a huge number of tuples. In order to prevent the browser from crashing or becoming unresponsive a limit needs to be applied. The full dataset matching a query is never considered if the

result is too large. At the time of this writing, the amount of measurements stored in the database are about to surpass 2 Million tuples. The client receives an unsummarized representation of the data and needs to do calculations in client-side JavaScript. This adds additional workload to the client who is already busy parsing JSON objects.
The chosen approach for the new implementation is to have separate queries for each visualization and summarize and aggregate the result as much as possible on the database level. The aim of this approach is to create lightweight JSON responses which can be easily handled by a client. This approach also allows to create summarized representations of the same underlying result set optimized for a given visualization. The approach shifts a lot of work away from the client to the database which is optimized for handling large amounts of data.
The goals for this version thus are:

1. **Data Retrieval**: Never retrieve more tuples than can be handled by the client (a few thousand at most). If a query returns a larger result set, then either only a partial view of the entire result set is provided to the client or further summarization must be done at the database level to reduce the tuple count.

2. **Aggregation**: Aggregate and reduce the data without distorting the original underlying dataset to create lightweight JSON responses from the server. The aggregated dataset must be representative for the original.

3. **Data Preparation**: Ideally, the client should not be handling any data formatting with procedural code. The client should optimally only have to deal with fetching and visualizing the data. The data should come pre-formatted from the server for the visualization at hand. The application design should thus adhere to the thin-client paradigm where the server-side has to do most of the work.

2 Application Architecture

The application is built using JavaScript to implement both front- and back-end logic. On the client side AngularJS v1.6.7 is used to dynamically manipulate and update the HTML. The server side implementation is using Node.js, a JavaScript runtime built on top of Chrome's V8 JavaScript engine written in C++ [1].

2.1 Overview

The AngularJS framework works by binding "controllers" to HTML nodes in the DOM. Once a controller is bound to a DOM element, all children elements can be manipulated dynamically by JavaScript defined in the associated controller logic. A controller is essentially a class definition bound directly to the DOM [2].
The TFDB front-end leverages this functionality by dividing the HTML DOM into several controllers. Fetching and updating the view for different parts of the HTML is thus handled by different controller classes. A short description of what each individual controller is responsible for is listed below:

- **FilterController**: Dynamically updates the filter parameters based on user input. Notifies all other controllers when the user clicks the search button.

- **TopQueriesController**: When the user clicks on a saved query, the saved query parameters are fetched from the database and passed to the FilterController.

- **TableController**: Renders and updates the table based on given filter parameters.

- **ChartController**: Handles the logic for all chart visualizations such as Scatter Chart, Box Plot and Violin Plot.

- **MapController**: Fetches and parses GeoJSON returned by the server and draws the result to Google Maps.

```
<html ng-app="feedbase">
  ...
<body>
  <div ng-controller="FilterController">
      ...
    <div ng-controller="TopQueries">
      <!-- List of Top Queries -->
    </div>
    <div ng-controller="ChartController">
      <!-- Chart markup -->
    </div>
    <div ng-controller="MapController">
      <!-- GoogleMaps markup -->
    </div>
    <div ng-controller="TableController">
      <!-- Table markup -->
    </div>

  </div>
</body>
</html>
```

Figure 1: AngularJS HTML controller Hierarchy Example.

In addition to that, a “FilterParams” object is used to store which feeds, nutrients years and geographical criteria a user has selected. This object is stored on the client and is used to build dynamic filter conditions on the server. More on this topic is described in section 2.2.2.

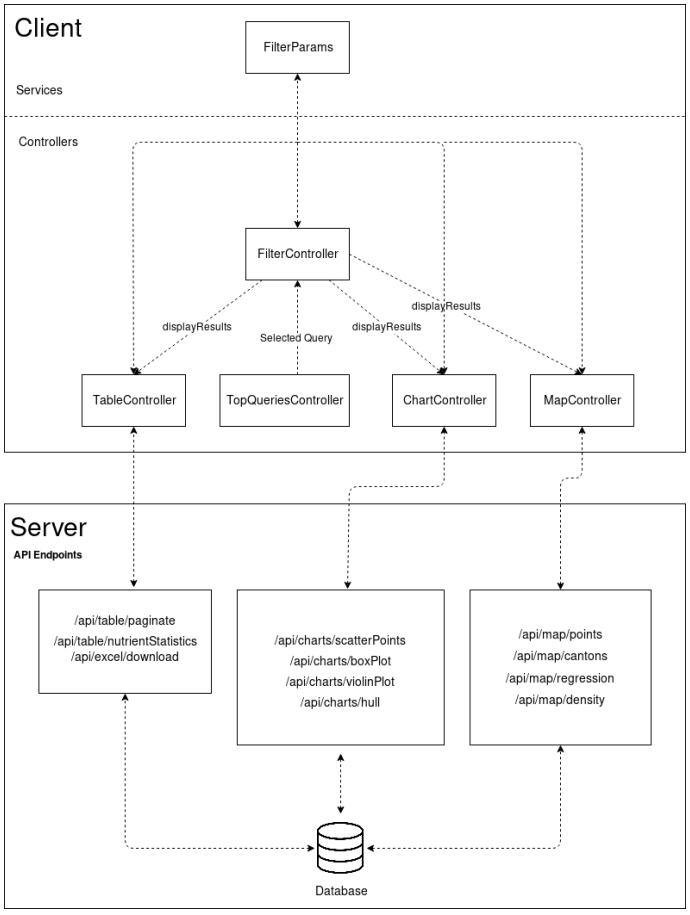

Figure 2: FeedBase Application Overview

The server-side API endpoints are used to serve any incoming HTTP requests made by the client. A detailed description of the endpoints can be found in appendix A.

2.2 Filtering & Query Building

The core of the application's functionality lies in allowing users to filter measurements based on feed type, nutrient, time and geographical conditions. The aim of filtering is to find all relevant facts that match the given set of conditions. A fact represents a measured quantity that can be associated with a feed, a time, a nutrient, an origin, a sample or an analyses method.

2.2.1 The FeedBase Star Schema

Information about a fact is stored in a fact table. The fact table contains the actual measurement value and multiple foreign keys pointing to tuples in relations that contain the associated information used in filters.

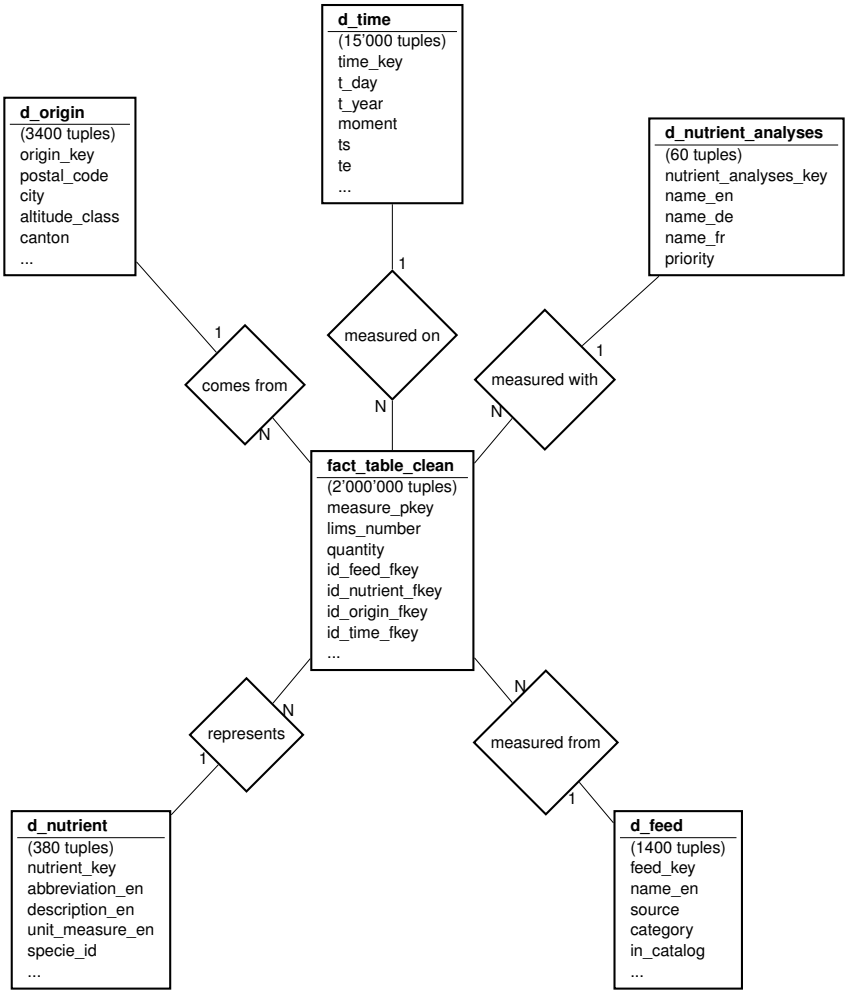

Figure 3: ER-Diagram of the relations involved in filtering with approximate tuple count

As depicted in figure 20, this results in a star schema with with one fact table and multiple dimension tables. The user can apply filters to all dimensions when querying the database. Thus, five consecutive joins on the fact table to all dimensions is required to get a base relation R on which a set of filter conditions C can be applied. This looks as follows:

$$R \leftarrow fact\_table \bowtie_{id\_feed\_fkey=feed\_key} d\_feed$$
$$\bowtie_{id\_nutrient\_fkey=nutrient\_key} d\_nutrient$$
$$\bowtie_{id\_origin\_fkey=origin\_key} d\_origin$$
$$\bowtie_{id\_time\_fkey=time\_key} d\_tie$$
$$\bowtie_{id\_analyses\_fkey=nutrient\_analyses\_key} d\_nutrient\_analyses$$
$$filtered \leftarrow \sigma_C(R)$$

The $filtered$ relation needs to be computed for every query and serves as a base to apply further aggregations, partitions and joins with other relations as needed by the visualization at hand.

2.2.2 Building Filter Conditions

The filter conditions C mentioned in 2.2.1 are derived from the tree selection checkboxes in the filter menu on the client. Each checkbox has an associated value (e.g., feed id, nutrient id, year, canton etc.) which is directly used in the WHERE statement of the resulting SQL query strings.
On the client side, these filter conditions are represented in a JSON object holding information about the user's selected checkboxes. This object represents an abstract description of the filter conditions C and is stored on the client-side. It is used to keep track of user input and is passed to the server on every request. The server builds the final WHERE clause used in all queries by ANDing the selected conditions. Using this implementation, the application can thus answer conjunctive queries where C is a set of conditions of the form $C = C_1 \wedge C_2 ... C_n$.

```
{
  language: 'de',
  loadlevel: 0,
  params: {
    agrideaFeeds: [1325],
    altitudes: ['> 1000'],
    cantons: ['Graubünden'],
    classFeeds: [],
    dataType: 'td',
    fresh: false,
    nutrients: [],
    selectedNutrient: null,
    selectedNutrients: [
                    180
                    144
                    158
                    163
                    160
                    159],
    selectedAnalyses: [],
    nutrientsDerived: [],
    raw: false,
    seasons: [],
    unclassFeeds: [],
    years: [2006],
    tablePage: 0,
    radius: 0,
    mapZoom: 7,
    mapNEBoundLat: 46.8448,
    mapNEBoundLng: 9.8677,
    mapSWBoundLat: 46.3731,
    mapSWBoundLng: 9.1261,
    location: null,
    selectedLocationLat: 46.8448,
    selectedLocationLng: 9.8677
  }
}
```

```
SELECT *
FROM filteredRelation
-- Selected Feeds and Nutrients
WHERE (id_feed_fkey IN (1325))
AND (id_nutrient_fkey IN (180, 144, 158,
163, 160, 159))
-- Analyses Filter
AND id_nutrient_analyses_fkey IN (...)
-- Geo Filter
AND d_origin.canton IN ('Graubünden')
-- Altitude Filter
AND d_origin.altitude_class IN ('>␣1000')
-- Time Filter
AND ((d_time.moment = 1 OR d_time.moment = 2)
            AND d_time.t_year IN (2006))
-- Map Bounds
AND (
d_origin.latitude <= 46.8448
d_origin.longitude <= 9.8677
d_origin.latitude >= 46.3731
d_origin.longitude >= 9.1261
)
-- point distance
AND
(ST_Distance(
 ST_MakePoint(9.8677,
 46.8448)::geography,
 ST_MakePoint(d_origin.longitude,
 d_origin.latitude)::geography
 ) < 20)
```

Figure 4: JSON Filter Object to SQL Mapping

This design works well for the specific use-case of building a dynamic WHERE clause based on user input whose conditions are conjunctive across the dimension tables and disjunctive for a selection within a dimension table. The JSON object can easily be extended with more key value pairs if new filter conditions are implemented. However, this mapping only works for an instance of the $filtered$ relation and the user is restricted to using the filter conditions that have a mapping to an SQL Where clause condition.
This functionality is provided by a 'WhereStatement' class that is implemented on the server-side. This class simply parses all keys from the given JSON object and builds an SQL query string that can be used in the WHERE clause of any query with the joined $filtered$ relation in the FROM clause.
The available filter options a user can select are organized in tree structures to denote categories and subcategories of feed types and nutrients in hierarchical order. The tree structures are stored in

the $vr\_classified\_feeds\_tree$ and $vs\_classified\_feeds\_tree$ relations for classified feeds and in the $vs\_classified\_nutrients$ relation for classified nutrients. These relations have a reflexive foreign key attribute to describe the tree structure.

vs_classified_feeds_tree

feed_group_id	parent_feed_group_id	feed_group_en
1	NULL	Raw material
2	1	Cereal grains
3	1	Cereal co-products
4	3	Milling
...	...	...

Table 2: Reflexive relation tree structure for classified feeds

2.3 Authentication

Authentication is implemented using JSON web tokens (JWT). A JWT is a token containing an encoded JSON object as payload. A decoded token's json payload has the following contents:

```
{
  exp: 1529053338,
  iat: 1528448538,
  userlevel: 2,
  username: 'someuser'
}
```

The exp key denotes date of expiry of the token. In this implementation the token expires after 7 days requiring the user to login again. The iat key denotes the date of issuance of the token. The $userlevel$ and $username$ key come from the $users$ table where the user level is used for authorization. A user level smaller than or equal to 3 are regular users which have different privileges. A user level above 3 are admin users. This information is stored as a cookie in the user's browser and passed to the server with every request. An encoded representation of the above payload as stored in the cookie looks as follows:

```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1MjkwNTMzMzgsImlhdCI6...
```

Note that a JWT payload is not encrypted but only encoded meaning that the JSON contained in the payload can be decoded by anyone. It should thus not contain sensible information. The tokens encoded payload representation however, is signed by a private/public RSA256 keypair stored on the server. This signature is checked by the server on incoming cookies and ensures that third-parties cannot login with a fake token payload as the signature would be invalid.
Authorization is implemented as a middleware function. A middleware function is a function that can be executed before or after a request is processed. In this use case, it is executed before the request is processed. The implementation provides a $checkUserLevel(requiredLevel)$ middleware which verifies the token and checks that the user level is at least the required level specified. This function can be included in a route definition to make certain actions only available the certain users. Below is an example from the route handling excel export of data.

```
router.post('/download',
 // Check if user level >= 8, reject otherwise
  local.checkUserLevel(8),
  // If user is authorized, execute this function
  function(req, res, next) {
        // actual download logic
  }
```

If at some point in the future any of these authorization checks must be adjusted to another user level it is a simple as adjusting the argument of the $checkUserLevel(requiredLevel)$ function.

3 Table Pagination

The approach in the current implementation was to pass a random selection of tuples from the $filtered$ base relation (from section 2.2.1) to the client and limit the result set size given a threshold value.
Queries whose result size were above the threshold value were thus limited and some tuples were not considered. The pagination of the resulting table was then done on the client-side.
Client-side table pagination i.e., fetching all results and creating the paginated table on the client works well for small result sets. However it is not a scalable approach and does not guarantee that the fetched results are representative.

3.1 Server-Side Table Pagination

The chosen approach to address the limitations of client-side table pagination is to push the pagination logic to the server. The client thus fetches only one table page at once. SQL provides LIMIT and OFFSET keywords which allow to easily implement this logic.
In this use-case, however, the $filtered$ relation represents individual nutrient values belonging to a particular measurement. The LIMS numbers, representing individual feed measurements, are thus in a vertical representation. In the final table visualization a LIMS number may only be listed once and the individual nutrient measurements (avg_quantity attributes) need to be horizontalized.

lims_number	id	an_id	avg_quantity	day	...
17-01763-001	144	7	899	2017-01-24	...
17-01763-001	163	2	201	2017-01-24	...
17-01763-001	158	2	101	2017-01-24	...
17-01510-002	158	2	67	2017-01-22	...
17-01510-002	163	2	82	2017-01-22	...
17-01510-002	144	7	933	2017-01-22	...

Table 3: Sample instance of $filtered$ relation

LIMS-Nr	Datum	PLZ	Kanton	Futtermittel	OS g/kg TS berechnet	RA g/kg TS NIRS	RA g/kg TS Thermogravimetrie (LECO 105°, 550°)	RP g/kg TS Dumas 6.25 (Verbrennung)	RP g/kg TS NIRS
17-01763-001	2017-1-24	2747	Bern	Hay 2.ff cut	899	101	101	201	201
17-01510-002	2017-1-22	6318	Zug	Hay 1. cut	933	67	67	82	82

Figure 5: Final table representation with horizontalized average quantities

The relation instance shown in table 3 must thus be mapped to the table representation in figure 5. The id attributes, representing a nutrient id, must thus be mapped to the correct column of the final table representation. As one can see from the relation instance in table 3, simple LIMIT and and OFFSET commands cannot be applied to this representation. Suppose a limit of 5 were applied. All quantities for LIMS number “17-01763-001” are kept but some quantities belonging to LIMS number “17-01510-002” are cut off.
The “last row” of the final table representation might thus be missing nutrient values. A simple approach to address this problem might be to count the number of nutrients a user has selected and set the limit accordingly. Suppose the user has selected n nutrients. If five rows should be displayed in the client, a limit of $5 * n$ could be applied. This approach, however, assumes that each LIMS number has an entry in the fact table for every nutrient. This is obviously not the case and the risk to cut off measurements remains.
A better approach is to “horizontalize” the $filtered$ relation such that the $lims\_number$ attribute is unique. A group by LIMS number must thus be achieved. The $filtered$ relation has attribute values which stay constant accross tuples with the same LIMS number (e.g., day) and has other attributes with varying attribute values (eg. id, an_id, $avg\_quantity$. In order not to loose the information of the varying attributes they are aggregated into arrays before grouping by LIMS number. The query to achieve this uses PostgreSQL’s $array\_agg()$:

```
SELECT lims_number,
array_agg(id) AS ids,
array_agg(an_id) AS an_ids,
array_agg(avg_quantity) AS avg_quantities
day
...
FROM filtered
GROUP BY lims_number, day
```

Figure 6: Array aggregation of attributes varying accross LIMS numbers

This results in the following representation:

lims_number	ids[]	an_ids[]	avg_quantities[]	day	...
17-01763-001	[144,163,158]	[7,2,2]	[899,201,101]	2017-01-24	...
17-01510-002	[158,163,144]	[2,2,7]	[67,82,933]	2017-01-22	...

Table 4: Sample instance of a "horizontalized" $filtered$ relation

The representation from table 4 allows to apply limit and offsets without losing information. The mapping from quantities to nutrients is also straightforward and can be done using the array indices.

3.2 Server-Side Sorting

The table on the client should be sortable i.e., when a user clicks on a column header the table should be sorted in ascending or descending order based on the selected column. With the horizontalized relation shown in section 3.1 this is not an easy task. If the user wants to sort by LIMS number or day or any other attribute staying constant accross tuples with the same LIMS number (see table 4) then this can be easily done with a simple ORDER BY.
If the user decides to sort by a nutrient column, the sorting task becomes more complex. Consider the relations shown in table 3 and table 4. The id attributes represent the nutrient. If the user wants to sort by the nutrient with an id of 144 in descending order, the final representation of the horizontalized relation should be the following:

lims_number	ids[]	an_ids[]	avg_quantities[]	day	...
17-01510-002	[158,163,144]	[2,2,7]	[67,82,933]	2017-01-22	...
17-01763-001	[144,163,158]	[7,2,2]	[899,201,101]	2017-01-24	...

Table 5: "Horizontalized" $filtered$ relation ordered by nutrient id 144 in descending order.

The LIMS number "17-01510-002" should be the top entry because the quantity 933 is the maximum value for nutrient id 144. The order of the array attributes does not matter and can be arbitrary. The array aggregated representation is not suited for ordering because the $ids[]$ and $avg\_quantities[]$ attributes are not atomic.
An order by nutrient essentially means that an order for a subset of the relation shown in table 3 must be established. In the final result relation, the $lims\_number$ attribute must be in the correct order given an order by over the $avg\_quantity$ attributes of a specific id - an_id combination. The ordering of the lims_numbers for a subset can be expressed as follows:

```
WITH orderedByNutrient AS (
  SELECT
  ROW_NUMBER() OVER (ORDER BY AVG(quantity) DESC),
  lims_number AS lims
  FROM filtered
  WHERE
  ...
  AND id_feed_fkey = 2
  AND id_nutrient_fkey = 163
  AND id_nutrient_analyses_fkey = 2
)
```

Figure 7: CTE representing sub query to order by a given nutrient

This CTE assigns a row_number to all relevant LIMS numbers where row_number is the final order for all LIMS numbers in the $filtered$ relation. In a next step, a join with the $filtered$ relation is required:

```
SELECT lims_number,
array_agg(id) AS ids,
array_agg(an_id) AS an_ids,
array_agg(avg_quantity) AS avg_quantities,
season,
canton,
postal_code,
day
FROM filtered
LEFT JOIN orderedByNutrient ON lims_number = orderedByNutrient.lims
GROUP BY lims_number, season, canton, postal_code, day, row_number
ORDER BY row_number
```

Figure 8: Left join with CTE ordered by nutrient value to create final ordering

A left join is used because not all LIMS numbers have measurements for all nutrient values and all tuples matching a given $filtered$ relation instance should be kept in the ordered representation.

3.3 Derived Nutrients Calculation

The table displaying reference data requires some nutrient values to be derived from other nutrient values of the same feed based on a formula. These formulas are stored in the $t\_formula\_feed$ and $t\_formula$ relation. In a first step, the formulas must thus be retrieved:

```
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 ...
```

Figure 9: Retrieving the formulas for derived nutrient calculation

This relation then contains the formula with placeholders whose values must be replaced by the nutrient value of the given nutrient id.

feed_key	nutrient_key	expanded_formula_eval	involved_nutrients_ids	correct
756	120	$_nv[163] * $_nv[121]	163,121	t
756	122	$_nv[163] * $_nv[123]	163,123	t
756	176	$_nv[163] * $_nv[177]	163,177	t
756	1	(0.9559 * $_nv[159]) + 24.461	159	t

Table 6: Formulas with their involved nutrients.

The values of the non-derived nutrients are then fetched from the joined $summary\_data$, $d\_nutrient$ and d_feed relation.

```
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,
  summary_data.raw_value AS raw_value,
  d_feed.feed_key AS feedkey,
  d_feed.name_de AS fname
  FROM
  summary_data,
  d_nutrient,
  d_feed
  WHERE ...
```

Figure 10: Retrieving non-derived nutrients

The nutrient values must now be mapped to the formulas such that the derived values can be calculated. This is done by creating an equivalent representation of the relation containing the formulas and of the relation containing actual nutrient values. A JSON object is built for each nutrient of the form:

```
{
  "involved_nutrients": [
    {
      "nutrient_id" : 159, "raw_value" : 120.894
    }
  ], "formula" : "(0.9559 *  $_nv[159] ) + 24.461"
}
```

Figure 11: JSON object containing a derived nutrient

This JSON representation is used for both derived and non-derived nutrients. The involved nutrients array has as many objects as there are placeholders in the formula. If the nutrient is not a derived nutrient, the formula key will be set to NULL and the involved nutrients array will contain one element (the value to be taken without any calculation). The SQL to build this JSON representation is depicted below:

```
SELECT
feed_key,
nutrient_key,
json_build_object(
'involved_nutrients', json_agg(
json_build_object(
  'nutrient_id',involved_id,
  'raw_value', raw_value
  )
),
 'formula',
 expanded_formula_eval)
...
```

```
SELECT
feedkey,
nid,
json_build_object(
'involved_nutrients', json_agg(
json_build_object(
        'nutrient_id',nid,
        'raw_value', raw_value
        )),
 'formula', NULL) as nutrient
...
```

Figure 12: Building an identical JSON representation for both actual and derived nutrients

In a last step the union of the two relations is built and grouped by individual feed. The JSON objects representing the nutrients are folded into an array.

```
SELECT feed_key, fname, json_agg(
        nutrient
) as nutrients
FROM (
SELECT * FROM formulasfolded
UNION ALL
SELECT * FROM rowsfolded
) as allNutrients
GROUP BY feed_key, fname
```

Figure 13: Uniting actual and derived nutrients

This leads to the following final representation:

feed_key	fname	nutrients
756	Hafer, Körner (lat. Avena sativa)	[{"involved_nutrients": ...}]
757	Hafer, Körner teilweise entspelzt (lat. Avena sativa)	[{"involved_nutrients": ...}]
...	...	...

Table 7: Final Representation for the reference data table

The nutrients attribute in table 7 now contains all derived and non-derived nutrients in an array of JSON objects. If an object has a formula, the server evaluates the formula before passing the nutrients to the client by replacing the placeholders from the representation seen in figure 11.

3.4 Range Search

The range search feature allows users to set a minimum and/or maximum value for each column in the table. A range can be defined e.g., by typing “25-30” to only get values within the range $25 \leq value \leq 30$. A minimum value without an upper bound can be set by e.g., typing “25” or “25-” which translates to $25 \leq value$ and a maximum value without a lower bound can be set by typing e.g., “-30” which translates to $value \leq 30$.

3.5 Evaluation

The pagination feature simply requires to fold some of the $filtered$ relation’s attributes into arrays. As can be seen from the query plan, this does not cause any significant additional cost:

```
Limit  (cost=49734.44..49758.69 rows=100 width=179)
  ->  GroupAggregate  (cost=49734.44..49943.05 rows=860 width=179)
        Group Key: rows.day,
        rows.lims_number,
        rows.season,
        rows.canton,
        rows.postal_code
        ->  Sort  (cost=49734.44..49755.94 rows=8603 width=99)
              Sort Key: rows.day DESC,
              rows.lims_number,
              rows.season,
              rows.canton,
              rows.postal_code
              ->  Subquery Scan on rows
              (cost=48699.04..49172.20 rows=8603 width=99)
                    ->  GroupAggregate
                    (cost=48699.04..49086.17 rows=8603 width=175)
                    ...
```

Figure 14: Table Pagination Query Plan

The last group aggregate at the bottom shows the cost for joining the fact table with all dimension tables. It almost makes up the total query cost.
Sorting by nutrient, on the other hand, requires the $filtered$ relation to be accessed twice. Once for fetching all tuples matching the filter conditions and once to query and enumerate a subset of the $filtered$ relation matching a given nutrient. It also makes another join necessary.

```
Limit  (cost=76241.29..76243.42 rows=50 width=187)
  CTE orderedbynutrient
    ->  WindowAgg  (cost=26438.58..26439.90 rows=75 width=27)
  ->  GroupAggregate  (cost=49801.40..50167.02 rows=8603 width=187)
        Group Key: orderedbynutrient.row_number,
        fact_table_clean.lims_number,
        ...
        ->  Sort  (cost=49801.40..49822.90 rows=8603 width=107)
              Sort Key: orderedbynutrient.row_number, (...)
              ->  Hash Left Join
              (cost=48701.48..49239.16 rows=8603 width=107)
```

Figure 15: Table Ordering Query Plan

The query plan shows that building accessing $filtered$ relation twice causes an increase in total query cost whereas the hash left join used to build the final ordered relation does no add any significant increase compared to the query plan without ordering.
Figure 16 shows the performance of table ordering and table pagination over an increasing amount of tuples. The result size reflects the number of tuples in the given instance of the $filtered$ relation before array aggregation.

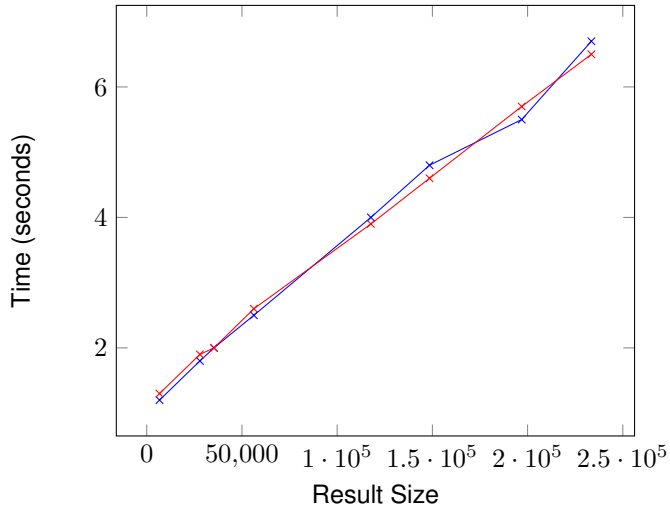

Figure 16: Performance of table ordering by nutrient (blue) and table pagination (red) with growing result size

4 Map Visualization

The map is used to visualize facts with a geo-reference. The user can toggle between showing map markers with the origin of a fact, a heatmap overlay denoting sample density and a heatmap showing the regression of a nutrient value.

4.1 Goal

The current implementation of the map simply passes all tuples from the $filtered$ relation (section 2.2.1) to Google Maps which then displays the latitude and longitude attribute as markers. This works fine for a few tuples but, as one might imagine, is not scalable as the amount of tuples grow. A location on the map might also reference many facts in a given query result. An example of this is shown in the relation below:

lims_number	id	an_id	latitude	longitude	...	feedname
05-15557188233176	163	2	47.34872400	8.11131600	...	Hay 1. cut
05-15557190233177	163	2	47.34872400	8.11131600	...	Hay 1. cut
05-15659241290243	163	2	46.53933700	9.23666500	...	Hay 1. cut
05-15670379474353	163	2	46.52253000	6.82771900	...	Hay 1. cut
05-15670379475353	163	2	46.52253000	6.82771900	...	Hay 1. cut
05-15670382478357	163	2	46.60115000	6.23921000	...	Hay 1. cut

Table 8: Sample instance of $filtered$ relation with geo-referenced tuples

This relation contains duplicate locations. There are cases in the FeedBase application where a location can represent thousands of facts. It does not make sense to serialize all those duplicate locations to JSON objects to send to the client. A significant amount of result tuples can be reduced by simply only retrieving distinct locations. As a first optimization the goal is thus to only pass relations of the following form to the map:

$$distinctLocations \leftarrow \Pi_{latitude,longitude}(filtered)$$

Figure 17: Projection to ensure no location is loaded twice on the map

Another goal in optimizing the location view is to optimize the case when many points are “close” to each other relative to the current zoom level of the map. The map does not provide a good overview of

the geographical distribution of the data when looking at areas densely populated with map markers.

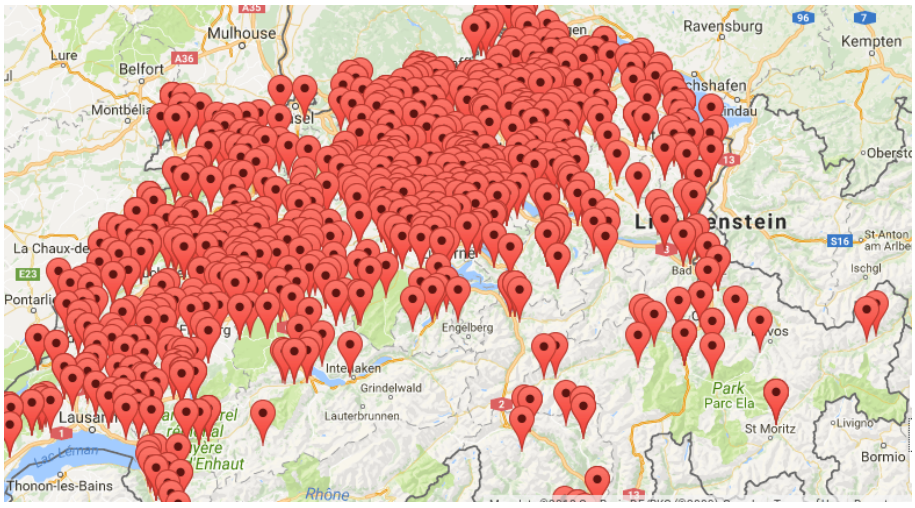

Figure 18: Map densely populated with map markers

The example visualization shown in figure 18 looks cluttered and does not provide a good overview of where the tuples actually are. It is not easy to tell at this zoom level in canton or region most of the markers are located.

4.2 Canton and Location View

In order to provide a good overview of the geo-referenced tuples at every zoom level of the map, the chosen approach is to abstract away the individual location markers and group them together to shapes. The canton geometries provide good and intuitive shapes to group markers at higher zoom levels.

4.2.1 Canton Shapes

The commune (Gemeinden) shapes of entire Switzerland were already stored in the database such that the canton shapes could be derived by creating the union of the commune shapes belonging to the same canton. Since the union of 2D-shapes in PostGIS is an expensive operation (takes about 6 seconds if run over all communes in Switzerland), a new relation $d\_cantons$ was added to the database storing the result of the union. The SQL used to achieve this is as follows:

```
CREATE TABLE d_cantons AS
  -- Derive the table from d_geometry and d_gemeinden
SELECT ST_Union(the_geom), kanton_nr
FROM d_geometry, d_gemeinden WHERE gemnr = gem_nr
AND kanton_nr IS NOT NULL
GROUP BY kanton_nr;
ALTER TABLE d_cantons ADD PRIMARY KEY (kanton_nr);
```

Figure 19: Building the $d\_cantons$ relation

The geometry attributes are 2-dimensional PostGIS polygon geometries describing the shape of each canton. The resulting $d\_cantons$ relation looks as follows:

d_cantons

kanton_nr	geom	kanton_name
1	POLYGON((680285 229552,680258.84 229588 ...	Zürich
2	MULTIPOLYGON(((572923 193948,573027 193973,573077 ...	Luzern
3	MULTIPOLYGON(((671936.199999999 206195.600000001 ...	Bern
...	...	...

Table 9: The $d\_cantons$ relation

The $kanton\_nr$ attribute is the primary key of the relation and was added to the d_origin relation as a foreign key such that the canton shapes can quickly be retrieved given a set of origins.

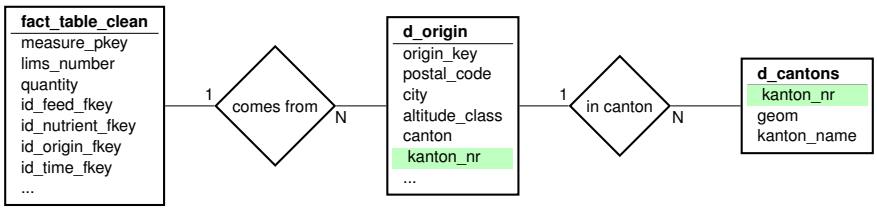

Figure 20: The $d\_cantons$ relation in the context of the fact table

The SQL implementation of the cantons query looks as follows:

```
/* retrieve all relevant canton numbers */
WITH origins AS (
      SELECT
      d_origin.kanton_nr AS kanton_nr,
      COUNT(*)
      FROM R
      WHERE ...
      -- conditions
      GROUP BY d_origin.kanton_nr
    )

/* retrieve the canton shapes with a join on kanton_nr */
SELECT * FROM d_cantons NATURAL JOIN origins
```

Figure 21: Retrieving canton shapes based on matching origins

The canton geometries are then colored according to the number of tuples that lie within a canton resulting in the final visualization. As the shapes are quite detailed, the description of each polygon border is quite large. If a query has a result that matches all 26 cantons in Switzerland, the data retrieved from the server has a size of about 5.75 MB. This, however, should not cause an issue since the number of cantons is not expected to grow in the near future.

4.2.2 Location Markers

When the user zooms in on the map, the canton shapes are removed and the location markers are shown to the user. The number of locations retrieved from the server is reduced to the locations that are within visible map bounds. When the user shifts the map bounds by dragging the map view, the map markers are recomputed.
The zoom level at which to switch from the cantons to the locations view is hard-coded meaning that once the canton overlay is resolved into map markers, there can be arbitrarily many location tuples needing to be fetched from the database. There is no logic for checking how many tuples would be fetched and loaded into the map. It is thus still possible that a query returns too many markers. This could happen when the user decides use the map in fullscreen mode. Depending on the screen size

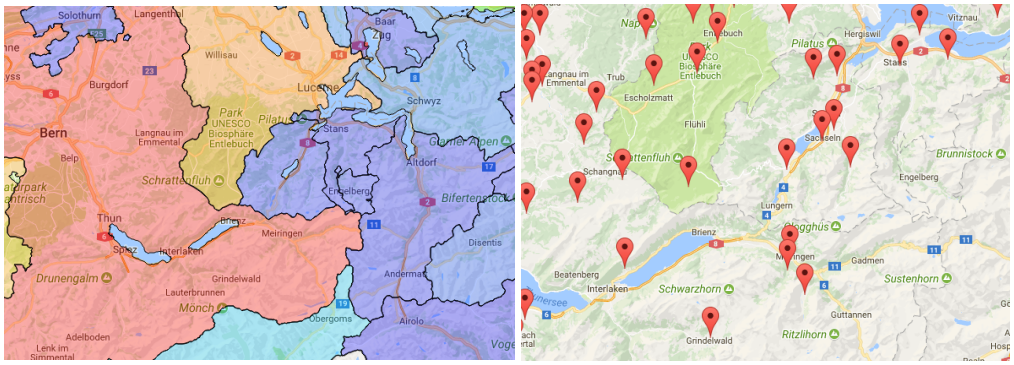

Figure 22: The canton shapes overlay (left) and the corresponding map markers when zooming in (right)

the map bounds can become large such that the database still may fetch too many tuples at once for the client.
Furthermore, the tuples representing the map markers are grouped by their $latitude$ and $longitude$ attributes. The $latitude$ and $longitude$ tuples thus need to have exactly matching numerical values. Even the smallest difference would cause to have multiple tuples appear at a seemingly same location.

4.3 Radius Search

The radius search feature allows the user to draw a circle shape around a map marker on the map to filter for results contained within the radius of the circle on the map. This feature filters all active visualizations (map, table and chart) adding the following condition to the WHERE clause of the respective queries:

```
ST_DWithin(
        ST_MakePoint(longitude, latitude)::geography,
        ST_MakePoint(7.889493000000016, 46.815685)::geography,
        19018.93569706483
)
```

The second argument of ST_DWithin is the center point of the circle and the third argument is the radius of the circle in meters. ST_DWithin returns true for points contained within the specified radius and false otherwise.

4.3.1 Event Propagation

The Google Maps API allows to bind to a $bounds\_changed$ event which triggers every time the circle radius is changed by the user. If this event is triggered, the MapController emits a $radiusSearch$ event to the FilterController which notifies all controllers to fetch data from the server with the new radius filter condition applied.

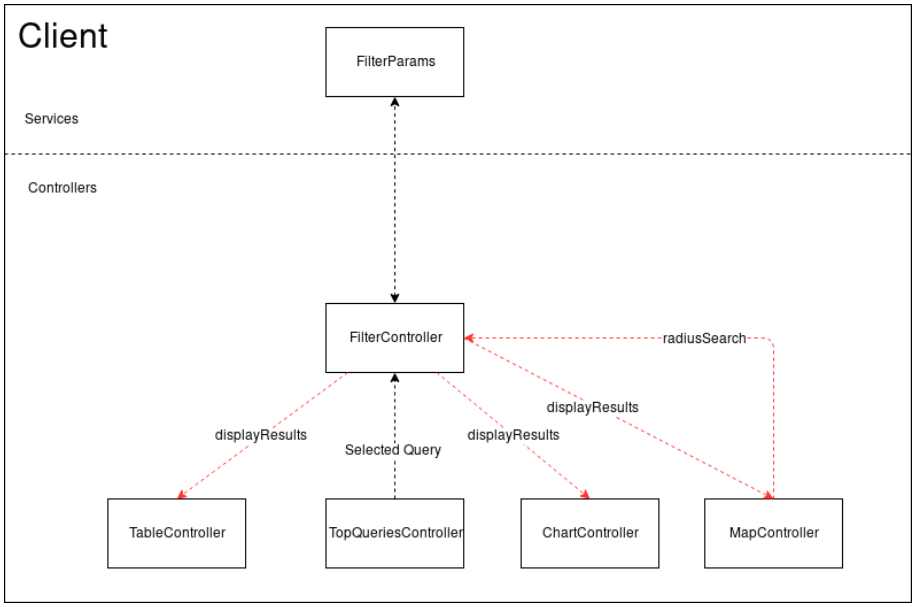

Figure 23: Client-Side Radius Search Event Propagation

This design allows the map to communicate with all active visualizations notifying them that the search radius changed. It provides a reactive response as the user drags the circle. Every time the user drags the circle, queries for all active visualizations are executed. Depending on the active visualization and the number of tuples contained in the specified radius, the response time to a radius search can become slow. A stress test has been carried out with the scatter chart, the table and the map over the entire hay dataset. The radius was set to 20 km for the first performance measurment incrementing until the whole area of Switzerland was contained in the radius. The results of this test are shown in figure 24.

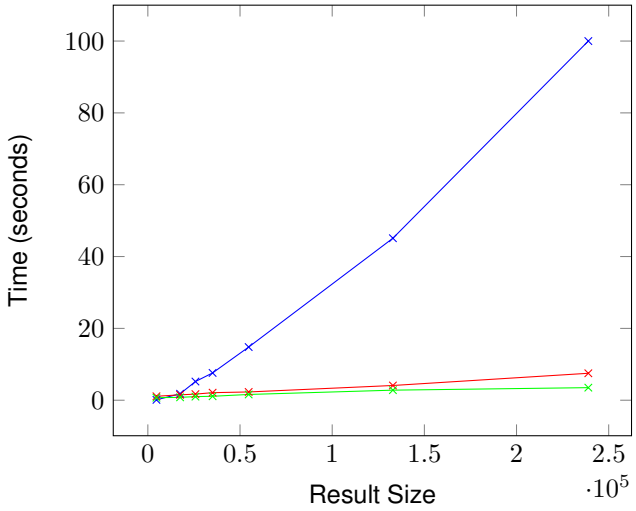

Figure 24: Radius Search response times of Scatter Chart (blue), table (red) and map (green) over a growing result size

The table and map response times stay relatively low over a growing result size. The scatter chart, however, takes almost two minutes to respond to the request for a summarization of around 250'000 tuples. The client was able to draw the chart immediately without any noticeable difference between.kgk

4.4 Sample Density and Nutrient Regression

The sample density and nutrient regression visualizations on the map are both visualized using the Google Maps heatmap feature. They only vary in their underlying query. The sample density query

simply computes a per location count for the given $filtered$ result set whereas the nutrient regression computes a per location quantity average for a given nutrient and analyses method.

```
SELECT
latitude,
longitude,
COUNT(*)::integer
FROM filtered
GROUP BY latitude, longitude
```

```
SELECT
latitude,
longitude
COUNT(*)::integer,
AVG(avg_quantity) AS quantity
FROM filtered
GROUP BY latitude, longitude
```

Figure 25: Sample density query (left) and regression query (right)

The heat map is then populated using weighted data points where each point is assigned a weight. The sample density overlay uses the count attribute as weight factor whereas the regression overlay uses the quantity attribute normalized to range $[0, 1]$.

```
var heatMapData = [
  {location: new google.maps.LatLng(37.782, -122.447), weight: 0.5},
  {location: new google.maps.LatLng(37.782, -122.443), weight: 2},
  {location: new google.maps.LatLng(37.782, -122.441), weight: 3}
]
```

Figure 26: Example definition of an array of weighted locations for the heat map [3]

4.4.1 Scaling the Heat Map

The heat map scales each point's radius according the current zoom level i.e., when zooming out a points radius is increased. This is due to Google Maps handling each point's radius relative to the map viewport by default. This may be desirable in some scenarios but, in this use case, leads to an unclear picture when zooming out from the map.
This issue can be solved by scaling the point radius according to the number of pixels within one meter. In order to compute pixels per meter for a given zoom level a mapping from world coordinates, referencing a point on the map uniquely, to pixel coordinates, referencing specific pixel on the map at a specific zoom level, is needed [4].

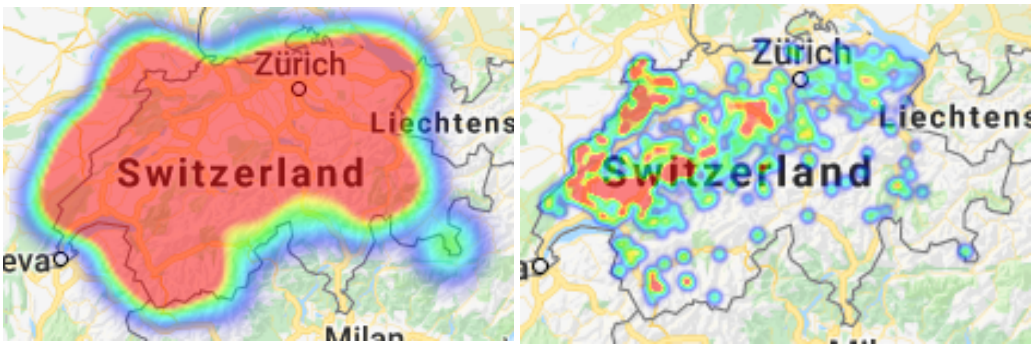

Figure 27: Heat Map with point scaling relative to map viewport (left) vs. pixels-per-meter scaling (right)

In order to calculate pixels per meter for the current zoom level, the center point of the map relative to the current viewport and a point 10 kilometers to the right of the center is calculated. These points are now available as latitude and longitude values.
In a next step, the center and offset point now need to be converted to pixel values. These pixel values represent the world coordinates. This is done by using a Mercator projection as implemented in

Google Maps. This projection is used to map the world's spherical surface to a cylinder such that it can be displayed on a flat surface. The Mercator projection for a given latitude and longitude is defined as:

$$x = \lambda - \lambda_0$$
$$y = \frac{1}{2} \ln \frac{1 + \sin(\phi)}{1 - \sin \phi}$$

Figure 28: Mercator Projection Formula [5].

where λ is the longitude and ϕ is the latitude. λ_0 is the longitude of the central meridian i.e., zero [5].

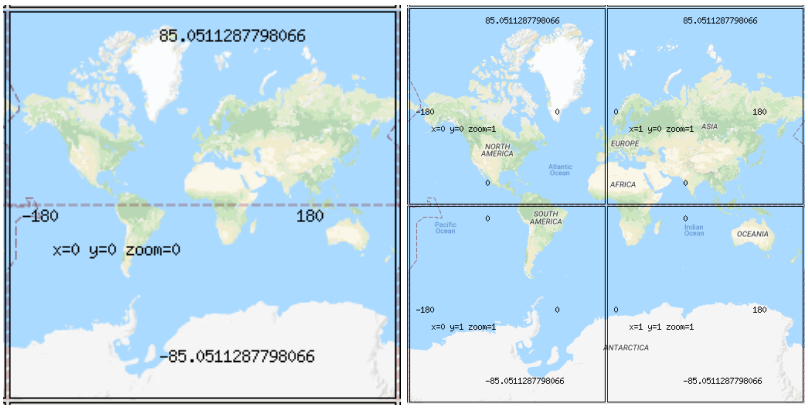

Figure 29: Google Maps Tile System with zoom level 0 (left) and zoom level 1 (right) [4].

Google Maps internally uses a tile system to load parts of the map relevant at the current zoom level. At zoom level 0 (fully zoomed out), the world coordinates are equal to the pixel coordinates. The origin of x and y axis is in the upper left. The available pixel space per tile is 256x256 pixels. At zoom level 1, the map consists of 4 256x256 pixel tiles resulting in a pixel space of 512x512 pixels [4].
For any given zoom level, each x and y pixel thus can be referenced with a value from $0 - 256^{zoomLevel}$ [4]. For a given world coordinate, it's corresponding pixel coordinate can be calculated with [4]:

$$pixelCoordinate = worldCoordinate * 2^{zoomLevel}$$

This property is used to compute the pixel coordinates for the center and offset point defined above. The absolute difference between those points then returns the number of pixels in one kilometer which can be used to derive the number of pixels in one meter. In the current heat map implementation each point is given a radius of 10 km. To make the heat map appear constant, every time the users zooms in or out on the map the radius is computed using $newRadius = 10000 * pixelsPerMeter$. A pseudo-code example of how radius calculation is implemented using the projection from figure 28 can be in figure 30.

```
GMAPS_TILE_SIZE = 256
pixelsPerLonDegree = GMAPS_TILE_SIZE / 360
pixelsPerLonRadian = GMAPS_TILE_SIZE / (2 * PI)

Function fromLatLngToPoint(latlng: LatLng) : Point
    point = new Point(0,0)
    origin = new Point(GMAPS_TILE_SIZE / 2,
      GMAPS_TILE_SIZE / 2)

    point.x = origin.x + latLng.lng() * pixelsPerLonDegree

    // Truncating to 0.9999 effectively limits latitude to 89.189.
    // This is about a third of a tile past the edge of the world tile.
    const siny = bound(sin(degreesToRadians(latLng.lat())),
      -0.9999, 0.9999)

    point.y = origin.y + 0.5 * Math.log((1 + siny) / (1 - siny)) -
      pixelsPerLonRadian
    return point
end
Function getNewRadius(zoomlevel: int, mapViewPortCenter:
  LatLng) : float
    numTiles = 2^zoomlevel

    // Get Point 10km to the right of the map view port center
    moved = computeOffset(center, 10000, 90)

    initCoord = fromLatLngToPoint(center)
    endCoord = fromLatLngToPoint(moved)

    initPoint = new Point(
        initCoord.x * numTiles,
        initCoord.y * numTiles
    )

    endPoint = new Point(
        endCoord.x * numTiles,
        endCoord.y * numTiles
    )

    pixelsPerMeter = abs(initPoint.x - initPoint.y) / 10000.0
    return floor(HEATMAP_POINT_RADIUS * pixelsPerMeter)
end
```

Figure 30: Pseudo-code for calculating pixel-per-meter ratio in Google Maps [6].

5 Scatter Chart Optimization

The Scatter Chart represents each fact as a point where the x-axis represents the time stamp when the fact was measured and the y-axis displays the average value for the selected measurement. Some queries return well over 30'000 points that render the client unresponsive. This section examines the possibilities for reducing the amount of data points passed to the client whilst maintaining a representative picture of the point distribution.

5.1 Goal

The goal is to group data points where they are “close” to each other. “Close” refers to the picture one gets in the final visualization of the points. If the points are in a range such that there are no visible gaps between them and one could draw a polygon over them without distorting the original (unsummarized) picture, then these points are candidates for summarization.

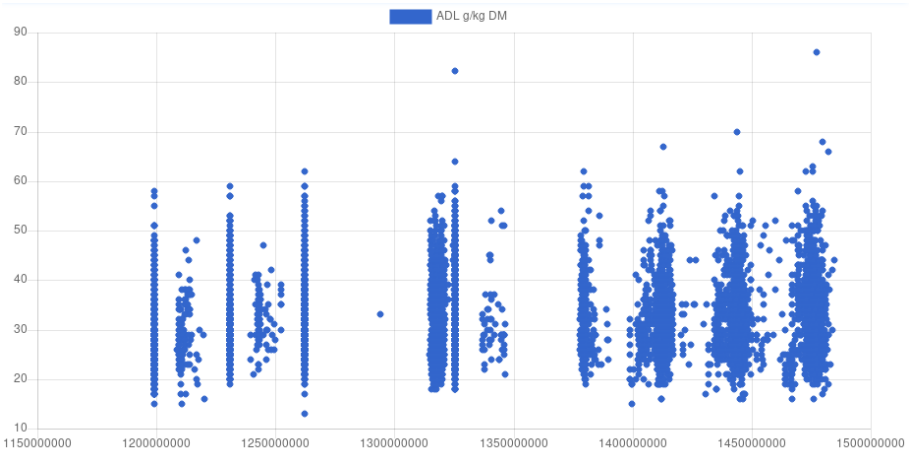

Figure 31: Sample Scatter Chart with several densely populated areas

The summarization approaches described in section 5.2 try to identify the points close enough, group these points using a proximity-based clustering algorithm and then keep only those points needed to describe the edges of the polygon.

5.2 PostGIS Clustering and Grouping

In order to achieve the visualization result described in 5.1 the overall approach used in the application can be broken down to these steps:

1. Retrieve all relevant points
2. Group the points by creating proximity-based clusters
3. For each group of points a convex/concave hull is created keeping only the points on the edge of a “densely populated” area.

PostGIS provides useful clustering and grouping functions out of the box that can be leveraged not only for the manipulation of geographical data. In fact any kind of points (and other geometries as well) having an extent in 2D/3D space can be manipulated with PostGIS.

5.2.1 Proximity Clustering

In order to group points that lie within a certain distance to each other PostGIS provides within distance clustering with the $ST\_ClusterWithin(geometry, distance)$ function. This function aggregates geometries to groups separated by no more than a specified distance. This works out of the box with geographical coordinates or if the x and y-axis of the chart have equal scales. In this use case however, the x-axis is a time scale and the y-axis shows nutrient values. The function only allows to specify one distance parameter which will apply to all dimensions. This issue can be addressed by normalizing the points x- and y-values to the range [0,1] before clustering. This is done using the following formula for each point’s x- and y-value:

$$x_i = \frac{x_i - min(x)}{max(x) - min(x)}$$

$$y_i = \frac{y_i - min(y)}{max(y) - min(y)}$$

This solves the problem of setting a clustering distance threshold that works well for both dimensions but the dimensionality of the x and y axes are lost. Section 5.2.3 shows how the normalized points are mapped back to the actual point values.

5.2.2 Convex vs. Concave Hull

The convex/concave hull algorithms are useful to compute polygons characterizing the area over a set of points. This is natively supported in PostGIS with the $ST\_ConvexHull$ and $ST\_ConcaveHull$ functions. The $ST\_ConvexHull$ function returns a convex shape given a set of point geometries whereas the $ST\_ConcaveHull$ may return non-convex as well as convex shapes.
A convex polygon is a subset S of the plane given that for any pair of points $p, q \in S$ the line segment $\overline{pq}$ is completely contained in S [7].

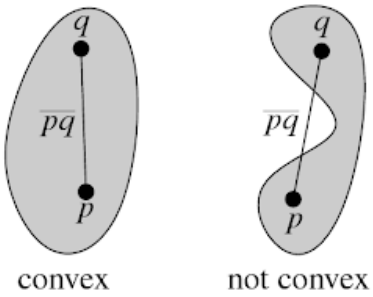

Figure 32: Convex and non-convex shapes given points p and q [7]

As one might see immediately in figure 32 there are infinitely many possibilities to form convex shapes given a set of points. The convex hull, on the other hand, identifies a unique shape for a given set of points.
This is due to the fact that the convex hull computes a convex polygon over a given set of points by connecting the outermost points. These can be connected to satisfy convexity. This is achieved by computing the set of outermost points P' given a set of points $P = \{p_1, p_2, p_3, ..., p_n\}$ such that $P' \subseteq P$.

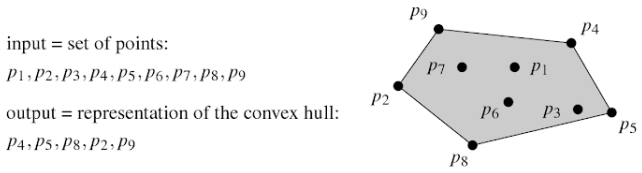

Figure 33: Computing a convex hull [7]

In some point distributions, a convex shape may work well to characterize the shape of the underlying points. However, the convexity constraint can result in distorting the underlying distribution. A good example to show when a convex hull will not work well is illustrated with the “C-Shape” distribution in figure 34.

Figure 34: “C-shape” distribution approximated with Convex Hull (left) and Concave Hull (right) [8].

In a distribution as shown in figure 34, a better shape overlay can be achieved with a concave hull. In the PostGIS implementation, concave hulls are computed by first calculating the convex hull and then approximating a concave hull by trying to reduce the area of the convex shape to a specified percentage. For example, if $ST\_ConcaveHull(geometries, 0.90)$ is called, PostGIS will compute a convex hull and then tries to reduce the area of the hull to 90% of the convex hull before exiting. This parameter is called the “target percent” to which to reduce the convex hull shape.

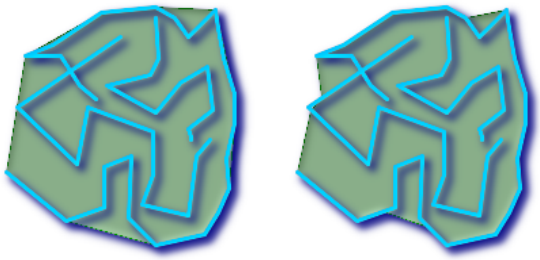

Figure 35: $ST\_ConcaveHull$ over a multiline with 100% target percent (left) and 99% target percent (right) [9]

In this implementation, the “target percent” parameter was set to 90%. This reasoning behind this choice is that lower percentages decrease the performance of the query significantly. Table 10 shows the performance of the concave hull function over 37’000 nutrient point tuples with a varying target percentage.

Target Percent Hull	t_{db}	n_{points}
100% (Convex Hull)	2.3s	37’000
95%	3.0s	37’000
90%	3.2s	37’000
80%	7.6s	37’000
70%	21.1s	37’000

Table 10: $ST\_ConcaveHull$ performance with decreasing target percent

5.2.3 Query Implementation: Cluster and normalize the points and map back to actual point values

Distance clustering on two independent dimensions may distort the picture significantly while normalizing the dimension between [0,1] works well to create convincing clusters. However, if normalized point values are used the information on the x- and y-axes of the chart are lost which renders the chart useless for interpretation.
In order to address these issues another possible approach is to use the normalized point values to create clusters, map the normalized points back to the actual points and, using this mapping, create clusters with the actual point values.
First all relevant nutrient measurements are selected from the $filtered$ relation:

```
 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
   FROM filtered
),
```

Figure 36: Retrieving relevant nutrient measurements for scatter chart

Now that all relevant nutrient quantities with their day of measurement are retrieved. The quantities and dates are normalized to the range [0,1].

```
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))
) AS day_normalized,
(avg_quantity - min_quantity) / (max_quantity - min_quantity)
AS quantity_normalized
FROM nutrientStats
),
```

Figure 37: Create a mapping between normalized and unnormalized time and nutrient value dimensions

The $statsNormalized$ relation now provides a mapping between the actual and the normalized date and average quantity values.

statsNormalized

id	an_id	avg_quantity	day	day_normalized	quantity_normalized
163	2	114	"2013-10-28"	0.731607629427793	0.634020618556701
163	2	106	"2013-10-28"	0.731607629427793	0.675257731958763
163	2	85	"2013-10-28"	0.731607629427793	0.783505154639175
163	2	90	"2013-10-28"	0.731607629427793	0.757731958762887
...	...	...	...	...	...

Table 11: Sample instance of the $statsNormalized$ relation

In the next step proximity clusters are created using the normalized values. The points within these clusters are then reduced to the points needed to create a concave hull:

```
-- 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
),
```

Figure 38: Cluster and apply concave hull to normalized point values

The hulls are now available as geometries but the dimensions of the x- and y-axes are lost:

clustered

id	an_id	geometry
163	2	LINESTRING(0.7223 0.8814,0.72546 0.8814)
163	2	POLYGON((0.7347 0.8853,...))
163	2	LINESTRING(0.72455 0.8144,0.7309 0.8144)
163	2	POINT(0.7273 0.9587)
...	...	...

Table 12: Sample instance of the $clustered$ relation

The $clustered$ relation now contains geometries in normalized 2D-space. These geometries now need to be mapped back to the unnormalized values. This is done by first enumerating the clusters and then splitting the polygon and linestring geometries up to the points describing their contour:

```
-- 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
)
```

Figure 39: Enumerate clusters and retrieve points describing the geometry envelopes

The enumeration of the geometries now provides the information which point belongs to which geometry.

geometryPoints

geom_number	id	an_id	dp
5	163	2	POINT(0 0.568208608247423)
6	163	2	POINT(0 0.580285670103093)
6	163	2	POINT(0 0.586622525773196)
7	163	2	POINT(0 0.602443659793814)
7	163	2	POINT(0 0.615248453608247)
8	163	2	POINT(0 0.62071881443299)
...	...	...	...

Table 13: Sample instance of the $geometryPoints$ relation

The last step towards retrieving the geometries with their actual values is to join the $geometryPoints$ relation with the $statsNormalized$ relation on their nutrient id's, chemical analyses id's and their normalized point values.

```
geometries AS (
  SELECT id, an_id,
  -- 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
)
```

Figure 40: Building the final “unnormalized” geometries

Given the mapping from normalized to actual values in $statsNormalized$ relation, points containing the actual values can be derived and collected into their corresponding geometries. For polygons, right-hand ordering of the points is applied such that the client-side code can simply follow the polygon points and fill the area.

geometries

id	an_id	geometry
163	2	MULTIPOINT(1466978400000 555)
163	2	LINESTRING(1466978400000 540,1467064800000 537)
163	2	POLYGON((1466978400000 529,...))
...	...	...

Table 14: Geometries with actual values

The $geometries$ relation now contains the wanted geometries. The final formatting passed to the client is built by creating an array of JSON objects with a group aggregate over the distinct nutrient and analyses keys:

```
-- 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 geometries
        ) as withCenters
) as json
GROUP BY id, an_id
```

Figure 41: Final Scatter Plot JSON representation for the client

id	an_id	geometries
132	2	[{"geometry" : {"type":"MultiPoint","coordinates":[[1451602800000,607]]}}]
145	11	[{"geometry" : {"type":"MultiPoint","coordinates":[[1451602800000,3]]}}]
165	37	[{"geometry" : {"type":"MultiPoint","coordinates":[[1451602800000,1.5]]}}]
...	...	...

Table 15: Geometry aggregation to JSON arrays

Table 15 represents the final representation for the client. Each tuple contains all geometries associated with a nutrient and analyses method.

5.3 Clockwise Ordering of Points

The client receives the point coordinates from the representation seen in table 15 without any ordering. This can cause distortions in the final drawing process since the client connects all points from polygon-type geometries in the order it receives the coordinates from the server before filling them with color. The client thus has to take care of ordering the points before drawing them onto the chart canvas. This is implemented by sorting the points in clockwise order around the center of the polygon.

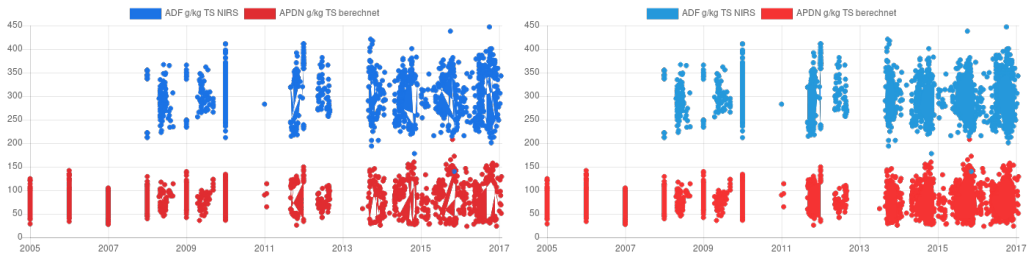

Figure 42: The same Scatter Chart without clockwise point ordering (left) vs. clockwise ordered polygons (right)

The center of the polygon is computed by finding the minimum and maximum x and y values of the points describing the polygon envelope. The minimum and maximum values are retrieved by first sorting the points from top to bottom along the y-axes and then taking the first and last element of the sorted array, and then sorting the points from left to right along the x-axes and taking the first and last

element as well. This yields the minimum and maximum values of both dimensions. The center is then retrieved as the average x and average y value as follows:

$$avg(x) = \frac{min(x) + max(x)}{2}$$
$$avg(y) = \frac{min(y) + max(y)}{2}$$
$$center = (avg(x), avg(y))$$

In a next step, the angle for each point with respect to the center point is calculated. This calculation is done by using the four-quadrant inverse tangent $atan2(y, x)$ function over the distance from each point to the center.
The order is established starting from the leftmost point on the x-axes. This point is already known since the points array has been sorted from left to right. The angle between the first point and the center is thus set as the start angle. This angle value is assigned to the left most point in the plane. The left most point should be the first entry in the points array after clockwise ordering. It is thus necessary that the numeric angle value assigned to this point is the smallest value if a sorting by angle value to the center should be applied. In an iteration over all points their angle values are calculated and assigned to the point. If an angle happens to be smaller than the start angle, its value is multiplied by 2π to make it greater than the starting angle numeric value which effectively results in a complete turn around the origin without changing its angle. For points whose angle is equal to greater than the starting angle. The angle value is assigned to the point without further calculations. Once every point has an angle value assigned with the start angle being the smallest value, the clockwise order can be established by sorting the array of points by their angle values in ascending order.
A pseudo-code description of this algorithm is shown below:

```
Function orderCW(points: Point[]): Point[]
    points = sort points array in descending y value order
    // Get the center point of the y-axes
    cy = (points[0].y + points[points.size - 1].y) / 2

    points = sort points array in ascending x order
    // Get the center point of the x-axes
    cx = (points[0].x + points[points.size - 1].x) / 2

    center = new Point(cx, cy)
    startAngle = NULL
    for point in points do
        angle = atan2(point.y - center.y, point.x - center.x)
        if startAngle == NULL then
            startAngle = angle
        else if angle < startAngle then
            angle += π * 2
        // Add angle value to point
        point.angle = angle
    end
    // Sorting points by angle results in clockwise ordering
    points = sort points in ascending angle value order
    return points
end
```

Figure 43: Clockwise sorting algorithm [10]

This algorithm works well with convex polygons if the center point is contained within the polygon. It may fail to establish the correct sort order in some instances if the server returns non-convex polygons where the center point is not contained within the polygon.

5.4 Evaluation

An implementation to reduce the amount of points needed in a scatter chart whilst still maintaining the original picture has been explored. The approach described in 5.2.3 seems to be viable because it allows to create proximity clusters in the normalized 2D-space without losing the original dimensions of the scatter chart's x- and y-axis.

```
GroupAggregate  (cost=87757.25..87762.29 rows=1 width=40)
  Group Key: geometrieswithcenters.id, geometrieswithcenters.an_id
  CTE nutrients
    -> GroupAggregate  (cost=51526.43..51526.93 rows=11 width=99)
    ...
  CTE nutrientstats
    -> CTE Scan on nutrients nutrients_4
    (cost=1.03..1.25 rows=11 width=44)
          ...
  CTE statsnormalized
    -> CTE Scan on nutrientstats  (cost=0.00..0.71 rows=11 width=36)
    ...
  CTE clustered
    -> HashAggregate  (cost=0.33..8.69 rows=1100 width=40)
          Group Key: statsnormalized.id, statsnormalized.an_id
          -> CTE Scan on statsnormalized
          (cost=0.00..0.22 rows=11 width=24)
  CTE clusteredenumerated
    -> WindowAgg  (cost=77.57..96.82 rows=1100 width=48)
   ...
  CTE geometrypoints
    -> CTE Scan on clusteredenumerated
    (cost=0.00..5791.50
    rows=1100000 width=48)
  ...
  CTE geometries
    -> GroupAggregate  (cost=30331.18..30331.32 rows=1 width=48)
          Group Key: points.id, points.an_id, points.geom_number
          -> Sort  (cost=30331.18..30331.18 rows=1 width=28)
                Sort Key: points.id, points.an_id, points.geom_number
                -> Hash Join  (cost=0.39..30331.17 rows=1 width=28)
                      Hash Cond: ((points.id = stats.id)
                      AND (points.an_id = stats.an_id))
                      Join Filter: ((points.dp ~=
                      st_makepoint(stats.day_normalized,
                      stats.quantity_normalized))
                      AND _st_equals(points.dp,
                      st_makepoint(stats.day_normalized,
                      stats.quantity_normalized)))
                      -> CTE Scan on geometrypoints points
                      (cost=0.00..22000.00 rows=1100000 width=48)
                      -> Hash  (cost=0.22..0.22 rows=11 width=36)
                            -> CTE Scan on statsnormalized stats
                            (cost=0.00..0.22 rows=11 width=36)
  -> Sort  (cost=0.03..0.04 rows=1 width=40)
        Sort Key: geometries.id, geometries.an_id
        -> CTE Scan on geometries
        (cost=0.00..0.02 rows=1 width=40)
```

Figure 44: Scatter Chart Query Plan

So how much additional cost does this normalizing and back-mapping to unnormalized point values cause? The query computes 7 common table expressions as intermediate steps towards retrieving the final representation. A shortened version of the query plan can be found in figure 44.
The most costly part of the query is building the $nutrients$ CTE. This relation is equivalent to the $filtered$ relation and needs to be computed for every query using the fact table. If no clustering were applied to the chart, the $nutrients$ CTE would suffice. The cost for building the $nutrients$ CTE is thus the cost for retrieving all nutrients without summarization. The cost for normalizing and clustering the points is relatively small and negligible. However, the back-mapping of normalized points to unnormalized points (in the $geometrieswithcenters$ CTE adds a significant amount of cost to the query. The join condition checking for equality between points remaining after concave hull reduction and the normalized values from the $statsNormalized$ CTE using “ST_Equals” is particularly expensive.
A performance evaluation of the query with increasing result size is shown in figure 45.

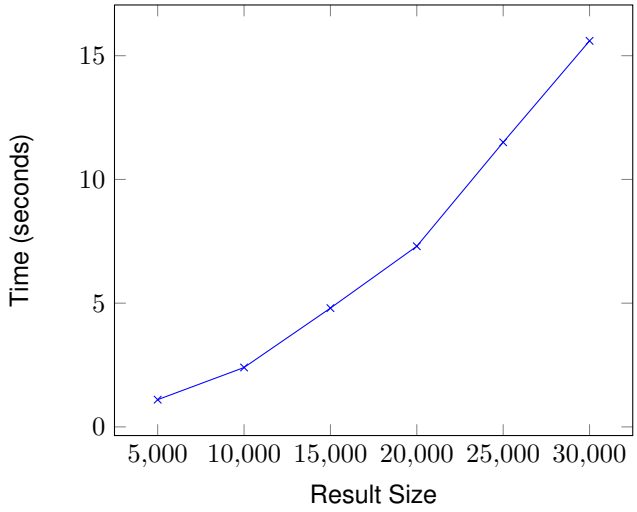

Figure 45: Performance of clustering of normalized values with back-mapping to unnormalized values

6 Correlated Nutrients Chart

The correlated nutrients chart is another kind of scatter chart showing how closely measurement values of two given nutrients correlate.

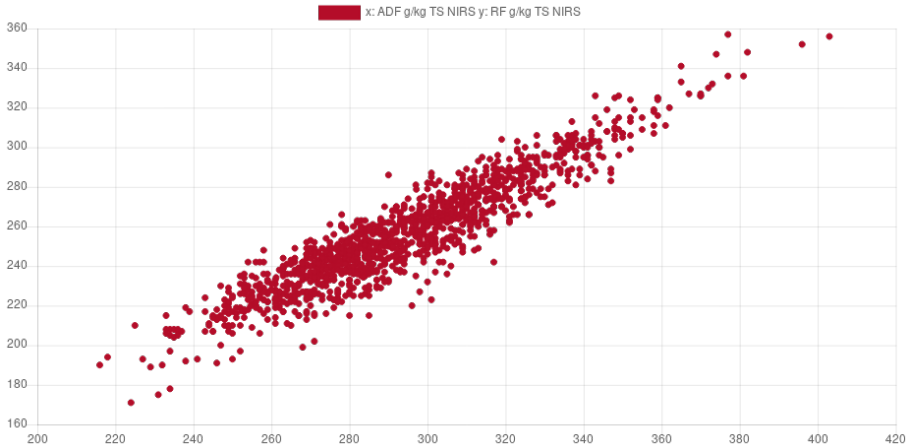

Figure 46: Sample correlation chart showing acid detergent fibre (ADF) on the x-axis and crude fibre (RF) on the y-axis

6.1 Query

The query works by creating two “data series” where the first series represents the nutrient measurements corresponding to the x-axis and the the second series the nutrient measurement corresponding to the y-axis. The two series are then simply joined on their LIMS number attributes to retrieve each points x and y values.

```
WITH series1 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
      FROM filtered1
      WHERE id_nutrient_fkey = 1
      AND id_nutrient_analyses_fkey = 2
    ),
    series2 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
      FROM filtered2
      WHERE id_nutrient_fkey = 2
      AND id_nutrient_analyses_fkey = 2
    )
-- Get the points to display in the scatter chart
SELECT s1.avg_quantity AS x,
s2.avg_quantity AS y
FROM series1 s1 JOIN series2 s2 ON s1.lims_number = s2.lims_number;

-- Compute the correlation coefficient
SELECT corr(s1.avg_quantity, s2.avg_quantity)
FROM series1 s1 JOIN series2 s2 ON s1.lims_number = s2.lims_number;
```

Figure 47: Correlated Nutrients Query

The above query is a simplified version of the actually implemented query with the purpose of only showing the important parts of the query. Further filters that apply in the respective WHERE clauses of the $series1$ and $series2$ relations have been omitted.
Currently the trendline of the correlated nutrients chart is still missing since the chart library in use does not provide an automated way to compute the line. Some way to derive the line using the correlation coefficient is being looked into.

6.2 Evaluation

In the current configuration of the query, no aggregation is applied in order to reduce the client-side load. However since this is just another form of scatter chart it is probably possible to apply the technique used for the scatter chart in section 5.2.3. This has not been implemented in this version.

7 Box Plot

The box plot is a visualization showing groups of nutrient measurement values through their quartiles. The client expects the min, $q1$, $median$, $q3$ and max values for each nutrient in order to display the

final boxplot where $q1$ and $q3$ denote the first quartile and the third quartile.

Figure 48: Sample boxplot chart comparis acid detergent fibre (ADF), crude ash (Ash) and Potassium (K)

7.1 Query

It is thus best to aggregate the measurements a relation of the form $boxplot(min, q1, median, q3, max)$ for each selected nutrient and return that to the client. To handle multiple boxplots in one query, aggregations over the partition by nutrient and analyses method are needed.

```
WITH tableRowsOrderedEnumerated AS (
SELECT
-- count row numbers and total number of records for each nutrient key
(ROW_NUMBER() OVER (
  PARTITION BY id_nutrient_fkey,
  id_nutrient_analyses_fkey
  ORDER BY AVG(quantity))) as row_number,
COUNT(*) OVER (
  PARTITION BY id_nutrient_fkey,
  id_nutrient_analyses_fkey
  ) AS total,
lims_number,
id_nutrient_fkey AS id,
id_nutrient_analyses_fkey AS an_id,
AVG(quantity) AS avg_quantity
FROM filtered
GROUP BY lims_number, id_nutrient_fkey, id_nutrient_analyses_fkey
ORDER BY avg_quantity
),
```

Figure 49: Retrieving relevant nutrient values for Box Plot

In a first step two additional attributes row_number and $total$ are computed. The row_number attribute denotes the order of the $avg\_quantity$ attributes within a nutrient a analyses partition (e.g, a row_number of 1 means that this is the minimum quantity for this nutrient and analyses method).

tableRowsOrderedEnumerated

row_number	total	lims_number	id	an_id	avg_quantity
1	20	330700-9	160	25	47.337616
1	538	16-4814	160	2	50.36726
2	20	327012-9	160	25	51.36204
2	538	16-4849	160	2	52.35602
3	538	16-1475	160	2	53.145336
...	...	...	...	...	...

Table 16: Nutrient and analyses method internal ordering with count of total samples

Using the representation shown in table 16 the min and max values can already be derived. However, the quartiles $q1$ and $q3$ and the median for each partition are still missing. The quartiles are computed by setting the tuple's attribute $q1$, $median$ or $q3$ equal to $avg\_quantity$ if the row_number meets one of the following conditions:

- $\lfloor \frac{total}{2} \rfloor \div 2 \leq row\_number \leq \lfloor \frac{total}{2} \rfloor \div 2 + 1$ for $q1$
- $\lfloor \frac{total}{2} \rfloor \leq row\_number \leq \lfloor \frac{total}{2} \rfloor + 1$ for the $median$
- $\lceil \frac{total}{2} \rceil + \lfloor \frac{total}{2} \div 2 \rfloor \leq row\_number \leq \lceil \frac{total}{2} \rceil + \lfloor \frac{total}{2} \rfloor + 1$ for $q3$

These conditions are expressed in SQL as follows:

```
quartiles AS (
  SELECT id, avg_quantity, row_number, an_id,
  AVG(CASE WHEN row_number >= (FLOOR(total/2.0)/2.0)
  AND row_number <= (FLOOR(total/2.0)/2.0) + 1
  THEN avg_quantity/1.0 ELSE NULL END
  ) OVER (PARTITION BY avg_quantity) AS q1,
  AVG(CASE WHEN row_number >= (total/2.0)
  AND row_number <= (total/2.0) + 1
  THEN avg_quantity/1.0 ELSE NULL END
  ) OVER (PARTITION BY avg_quantity) AS median,
  AVG(CASE WHEN row_number >= (CEIL(total/2.0) + (FLOOR(total/2.0)/2.0))
  AND row_number <= (CEIL(total/2.0) + (FLOOR(total/2.0)/2.0) + 1)
  THEN avg_quantity/1.0 ELSE NULL END
  ) OVER (PARTITION BY avg_quantity) AS q3
  FROM tableRowsOrderedEnumerated
)
```

Figure 50: Computing $q1$, $q2$ and $median$ values

A NULL value is set for attributes not matching any of the conditions. A sample representation of the $quartiles$ relation is structured as follows

quartiles

id	avg_quantity	row_number	an_id	q1	median	q3
160	57.12998	5	25	57.12998	NULL	NULL
160	58.237206	6	25	58.237206	NULL	NULL
...	...	...	...	...	...	...
160	62.47216	10	25	NULL	62.47216	NULL
160	62.485912	11	25	NULL	62.485912	NULL
...	...	...	...	...	...	...
160	87.02721	15	25	NULL	NULL	87.02721
160	90.28186	16	25	NULL	NULL	90.28186

Table 17: $q1$, $median$ and $q3$ value representation for an id - an_id combination

The final relation passed to the client is then computed with the following query:

```
SELECT
id,
an_id,
MIN(avg_quantity),
AVG(q1) AS q1,
AVG(median) AS median,
AVG(q3) AS q3,
MAX(avg_quantity) AS max
FROM quartiles
GROUP BY id, an_id;
```

Figure 51: Box Plot final aggregation

7.2 Evaluation

Each box plot can be represented by one tuple with 5 attributes (min, $q1$, $median$, $q3$, max) which is the minimal amount of information needed to draw a box plot. The final representation is thus optimally aggregated for the client. All goals formulated in 1.2 are met with this implementation. The client can easily handle loading many box plots at the same time.
The queries most costly part is building the filtered relation (i.e., executing all joins of the fact table with the dimension tables which makes up almost all the cost of the query) which is used in the "tableRowsOrderedEnumerated" CTE. The logic to compute the min, $q1$, $median$, $q3$ and max attributes comes almost for free.

```
HashAggregate  (cost=7006.69..7006.71 rows=1 width=48)
  Group Key: quartiles.id, quartiles.an_id
  CTE tablerowsorderedenumerated
    -> WindowAgg  (cost=7006.44..7006.49 rows=1 width=115)
         -> WindowAgg  (cost=7006.44..7006.47 rows=1 width=107)
              -> Sort  (cost=7006.44..7006.45 rows=1 width=99)
                   Sort Key: fact_table_clean.id_nutrient_fkey,
                   fact_table_clean.id_nutrient_analyses_fkey,
                   (avg(fact_table_clean.quantity))
                   -> Nested Loop  (cost=3880.80..7006.38 rows=1
                                                    width=147)
...
CTE quartiles
->  WindowAgg  (cost=0.03..0.17 rows=1 width=48)
      ->  Sort  (cost=0.03..0.04 rows=1 width=32)
            Sort Key: tablerowsorderedenumerated.avg_quantity
            ->  CTE Scan on tablerowsorderedenumerated
            (cost=0.00..0.02 rows=1 width=32)
->  CTE Scan on quartiles  (cost=0.00..0.02 rows=1 width=40)
```

Figure 52: Box Plot Query Plan

8 Nutrient Statistics

The nutrient statistics table provides a basic numerical overview (total count, average, minimum, maximum and standard deviation values) over the distribution of each nutrient and its associated analyses method.

8.1 Indicative Measurements

Usually the nutrient values are aggregated over the partition by their nutrient and analyses method. In case the query returns the same nutrient multiple times with varying analyses methods, an indicative aggregation should be carried out aggregating only over the partition by nutrient id. This way the user can see the distribution over all analyses methods.

Nutrients	Count	Average	Min	Max	σ
ADF g/kg DM NIRS	1267	293.618	216.000	403.000	28.042
ADL g/kg DM indicative	1209	36.643	18.000	59.000	8.101
ADL g/kg DM NIRS	514	35.364	18.000	59.000	7.446
ADL g/kg DM NA	695	37.588	18.000	59.000	8.435

Figure 53: Sample nutrient statistics table with ADL g/kg DM indicative as well as the individual analyses methods

8.2 Query

```
SELECT
id,
-- set 0 as analyses id to make ordering easier
0 AS an_id,
count,
max,
min,
avg,
stddev,
MIN(nutrient_name) AS nutrient_name,
MIN(unit_measure) AS unit_measure,
MIN(an_name) AS an_name
FROM (
  SELECT
  DISTINCT id, an_id,
  COUNT(*) OVER (PARTITION BY id),
  ROUND(MAX(avg_quantity) OVER (PARTITION BY id)::numeric, 3) AS max,
  ROUND(MIN(avg_quantity) OVER (PARTITION BY id)::numeric, 3) AS min,
  ROUND(AVG(avg_quantity) OVER (PARTITION BY id)::numeric, 3) AS avg,
  ROUND(
  COALESCE(
        STDDEV(avg_quantity) OVER (PARTITION BY id),
        0)::numeric, 3)
  AS stddev,
  nutrient_name,
  unit_measure,
  'indicative' AS an_name
  FROM filtered
) AS indicative
GROUP BY id, count, max, min, avg, stddev
-- Only get tuples that have more than one analyses method
HAVING COUNT(an_id) > 1
...
```

Figure 54: Nutrient Statistics Indicative Measurements Query

The final query is a union of two queries. The first one serves to retrieve indicative statistics and thus partitions the aggregation only by nutrient id. The HAVING clause at the end ensures that only indicative measurements are returned for nutrients actually having more than one associated analyses method in the given $filtered$ relation. These indicative statistics are then combined in a union with the aggregated statistics partitioned by nutrient and analyses method:

```
...
UNION -- union with indicative measurements
SELECT DISTINCT id, an_id,
COUNT(*) OVER (PARTITION BY id, an_id),
ROUND(MAX(avg_quantity) OVER (PARTITION BY id, an_id)::numeric, 3) AS max,
ROUND(MIN(avg_quantity) OVER (PARTITION BY id, an_id)::numeric, 3) AS min,
ROUND(AVG(avg_quantity) OVER (PARTITION BY id, an_id)::numeric, 3) AS avg,
ROUND(COALESCE(
        STDDEV(avg_quantity) OVER (PARTITION BY id, an_id), 0)::numeric, 3)
    AS stddev,
nutrient_name,
unit_measure,
an_name
FROM filtered
ORDER BY id, an_id
```

Figure 55: Union of indicative and non-indicative measurements

8.3 Evaluation

The query provides an optimally aggregated representation for the visualization at hand. Each result tuple can directly be represented in the statistics table. The amount of tuples returned depends on the cardinality of the selected grouping factors which have a multiplicative effect on the number of tuples in the result relation. Most available grouping factors such as season, canton, feed and altitude in meters are unlikely to grow quickly. The available temporal grouping factor “years” however is an ever increasing factor with time passing by and will cause the nutrient statistics table to grow.

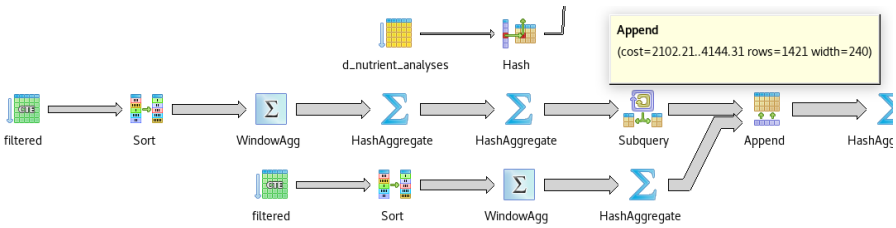

Figure 56: Cost spike when building the union from indicative and non-indicative measurements

Aside from the cost for building the $filtered$ relation, the union between indicative and non-indicative measurements causes the query cost to double since the almost the same subquery (windowed aggregations over slightly different partitions) is executed twice before uniting the two result sets. However, this is negligible since the since the cost for building the $filtered$ relation outweighs cost for aggregating nutrient statistics.

9 Conclusion

In this thesis, dedicated queries were implemented for each visualization of the Swiss Feed Database's web application with the aim of creating a stable and scalable solution for an ever-growing database. The utilization of spatial data types and functions for the map and scatter chart visualizations using PostGIS allowed to abstract away data points into shapes effectively reducing the amount of tuples the client has to process whilst maintaining a representative picture of the underlying distribution.
The data table has been reimplemented using server-side pagination and sorting allowing to navigate the table view over the entire dataset regardless the amount of tuples underlying the visualization.
The properties of the box plot and nutrient statistics visualizations allowed to create an optimally aggregated representation for the client at the database layer.
Although, the queries now always consider the full dataset without any limits applied, the evaluations have shown that, over a growing size of measurement data, the time needed for the database to produce a response increases leading to longer wait times for the client. Furthermore the violin plot and correlated nutrients visualization have not been optimized and receives unsummarized nutrient measurement tuples. These and further improvements to the application are left open to future works.

A API Endpoints

- **POST /api/table/paginate**
  Returns a json representation of rows and columns that are rendered in the table.
  Request Parameters: FilterParams
  Response:

```
{
  cols: [
  {
    "id":160,
    "an_id":29,
    "abbreviation":"RL",
    "unit":"g/kg TS",
    "an_name":"aggregiert"
  },
  ...
  ]
  rows: [
   {
     "lims_number":"xxx-3",
     "ids":[...],
     "an_ids":[...],
     "avg_quantities":[...],
     "feedname":"Gerste ...",
     "season":"Sommer",
     "canton":null,
     "postal_code":null,
     "day":"2017-6-8",
     "highlight":false
   },
   ...
  ]
}
```

- **POST /api/table/scatterPointQuery**
  Route to handle the case when a user clicks on a point in the scatter chart. Sets the highlight attribute to true for rows that match to the clicked scatter point. **Request Parameters**: FilterParams

```
{
  cols: [
  {
    "id":160,
    "an_id":29,
    "abbreviation":"RL",
    "unit":"g/kg TS",
    "an_name":"aggregiert"
  },
  ...
  ]
  rows: [
   {
     "lims_number":"xxx-3",
     "ids":[...],
     "an_ids":[...],
     "avg_quantities":[...],
     "feedname":"Gerste ...",
     "season":"Sommer",
     "canton":null,
     "postal_code":null,
     "day":"2017-6-8",
     "highlight":true
```

```
    },
    ...
  ]
}
```

- **POST /api/table/nutrientStatistics**
  Returns the min, max, avg and std. deviation values for a given set of nutrients
  Request Parameters: FilterParams
  Response:

```
[
   {
      "id":160,
      "an_id":27,
      "count":"3",
      "max":"461.425",
      "min":"454.112",
      "avg":"458.576",
      "stddev":"3.914",
      "nutrient_name":"RL",
      "unit_measure":"g/kg TS",
      "an_name":"PSE (HCl Aufschluss,...)"
   },
   ...
]
```

- **POST /api/charts/hull**
  Returns all measurements of selected nutrients across a selection of feeds, a given set of months/years and region. These measurement values are then passed to a scatter chart
  Request Parameters: FilterParams
  Response:

```
[
  {
    "id":284,
    "an_id":54,
    "geometries":[{
      "geometry":{
        "type":"LineString",
        "coordinates":[
          [ 1345500000000, 356.77887 ],
          [ 1345500000000, 355.97662 ]
        ]
      },
      "center":{
        "type":"Point",
        "coordinates":[ 1345500000000, 367.800245 ]
      }
     },
     ...
    ]
  }
  ...
]
```

- **POST /api/charts/boxPlot**
  Returns min, q1, avg, q3 and max values of selected nutrients across a selection of feeds, a given set of months/years and region. These measurement values are then passed to a box plot.
  Request Parameters: FilterParams
  Response:

```
[
   {
      "id":284,
      "an_id":54,
      "min":311.53775,
      "q1":336.283645,
      "median":355.97662,
      "q3":425.417495,
      "max":487.64474
   }
]
```


- **POST /api/charts/violinPlot**
  Returns all measurements of selected nutrients across a selection of feeds, a given set of months/years and region. These measurement values are then passed to a violin plot.
  Request Parameters: FilterParams
  Response:


```
[
   {
      "avg_quantities":[
         311.53775,
         ...
      ],
      "id":284,
      "an_id":54
   }
]
```


- **POST /api/map/points**
  Returns point geometries in GeoJson format where tuples for the given filter conditions are found
  Request Parameters: FilterParams
  Response:


```
{
   "type":"FeatureCollection",
   "features":[
      {
         "type":"Feature",
         "geometry":{
            "type":"Point",
            "coordinates":[
               8.353567,
               47.176361
            ]
         },
         "properties":{
            "f1":14,
            "f2":"Abtwil AG"
         }
      }
   ]
}
```


- **POST /api/map/cantons**
  Returns canton geometries in GeoJson format where tuples for the given filter conditions are found
  Request Parameters: FilterParams
  Response:


```
{
```

```
   "type":"FeatureCollection",
   "features":[
      {
         "type":"Feature",
         "geometry":{
            "type":"MultiPolygon",
            "coordinates":[
               [
                  [
                     [
                        8.5540853479946,
                        47.5531892401658
                     ],
                     ...
                  ]
               ]
            ]
         },
         "properties":{
            "f1":13,
            "f2":"Vaud",
            "quantity_normalized":1
         }
      }
   ]
}
```


- **POST /api/map/density**
  Returns the sample count per distinct location
  Request Parameters: FilterParams
  Response:


```
[
   {
      "latitude":"47.17636100",
      "longitude":"8.35356700",
      "count":42
   },
   {
      "latitude":"47.42287400",
      "longitude":"8.51473300",
      "count":93
   },
   ...
]
```


- **POST /api/map/regression**
  Returns the count per location and the average quantity of a selected nutrient measurement.
  Request Parameters: FilterParams
  Response:


```
[
   {
      "latitude":"47.42287400",
      "longitude":"8.51473300",
      "count":1,
      "quantity":57.913788,
      "quantity_normalized":1
   },
   {
      "latitude":"47.58950000",
      "longitude":"8.50068900",
      "count":18,
```

```
        "quantity":50.6160487222222,
        "quantity_normalized":0.28556525607192784
    }
    ...
]
```

- **GET /api/queries/**
  Returns the top saved queries
- **POST /api/queries/**
  Saves a new query to the database
  Request Parameters: FilterParams
- **GET /api/queries/:id**
  Returns the query parameters from a saved query
  Response:

```
{
    "dataType":"td",
    "agrideaFeeds":[

    ],
    "classFeeds":[
        "744"
    ],
    "unclassFeeds":[

    ],
    "nutrients":[
        "231",
        "132",
        "1",
        "180",
        "158",
        "163",
        "159"
    ],
    "nutrientsDerived":[

    ],
        ...
}
```

- **DELETE /api/queries/:id**
  Deletes the query with the given id
- **POST /api/login**
  Login a user given a username and a password
  Request Parameters:

```
{
  "username": "test",
  "password": "password123"
}
```

- **POST /api/logout**
  Logs a user out
- **POST /api/excel/download**
  Returns an excel sheet to the client with tuples matching the filter conditions.
  Request Parameters: FilterParams

B Technical Manual

For both development and production environment, PostgreSQL version 9.6 with PostGIS version 2.3 or above is required. The install commands below are for a Debian based Linux distribution and may vary across distros.

B.1 General Setup

1. With a running Postgresql and PostGIS setup, NodeJS version 8.11 or higher is required:

   ```
   curl -sL https://deb.nodesource.com/setup_8.x | sudo -E bash -
   sudo apt-get install -y nodejs
   ```

2. Verify that NodeJS has been installed:

   ```
   node -v
   ```

   It should e.g., say v8.11.2

3. In the project's root directory, a public/private RSA256 keypair is required for authentication. Create a keypair named jwtRS256 with the following command:

   ```
   ssh-keygen -t rsa -b 2048 -f jwtRS256.key
   # Don't add passphrase
   openssl rsa -in jwtRS256.key -pubout -outform PEM -out jwtRS256.key.pub
   ```

4. In the project's root directory, create a JSON file named params.json with the following contents:

   ```
   {
     "db": {
       "user": "your_db_user",
       "password": "your_db_password",
       "database": "tfdb",
       "host": "localhost",
       "port": 5432
     },
     "GMAPS_API_KEY": "your api key"
   }
   ```

   Set the parameters according to your setup

5. In the project's root directory, install all npm dependencies

   ```
   npm install
   ```

6. Run the application

   ```
   npm start
   ```

   If everything worked, the app should be running on localhost:3000

B.2 Setup for production

To keep the NodeJS server always up and running and let it start automatically on server reboot, an easy way is to run NodeJS as a Systemd Unit.

1. To create a new systemd unit navigate to /etc/systemd/system and create a new service file named feedbase.service with the following contents:

```
[Service]
ExecStart=/usr/bin/node /path/to/feedbase-version3/bin/www
Restart=always
# Only run feedbase if postgres is running
Requires=postgresql.service

StandardOutput=syslog
StandardError=syslog
SyslogIdentifier=feedbase-version3
# Adjust User and group to the user and group running the application
User=feedbase
Group=feedbase
# Run Node in production mode
Environment=NODE_ENV=production

[Install]
WantedBy=multi-user.target
```

2. The FeedBase service can now be started using the following command

```
sudo systemctl start feedbase.service
```

3. Verify that the service has started

```
systemctl status feedbase.service
```

The output should be similar to the following:

```
feedbase.service
   Loaded: loaded (/etc/systemd/system/feedbase.service; ...)
   Active: active (running) since Mon 2018-06-04 22:00:13 UTC; 2 days ago
```

4. Enable the service to keep the application always up and running

```
sudo systemctl enable feedbase.service
```

5. The application is now running on http://localhost:3000. To make the application run through a web server such as Apache or Nginx a proxy to port 80 or 443 (for https) is needed. Here the steps for Nginx are described. It should be similar for Apache.
To configure Nginx, open the /etc/nginx/nginx.conf file.

6. Make sure the following line is included in the http block (if not already there):

```
http {
  ...
  # Make sure this line is included
  include /etc/nginx/conf.d/*.conf;
}
```

This line is needed such that configuration files in /etc/nginx/conf.d/ are read.

7. Create a new file feedbase.conf in /etc/nginx/conf.d/feedbase.conf. Now the mapping between port 3000 where the application is running and port 80 is created.

```
upstream feedbase_server {
        server 127.0.0.1:3000;
        keepalive 8;
}

server {
  # 443 for https
```

```
    listen 80;
    # Adjust to url
    server_name vm-164.s3it.uzh.ch;

    location / {
      proxy_set_header X-Real-IP $remote_addr;
      proxy_set_header Connection "upgrade";
      proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
      proxy_set_header Host $http_host;
      proxy_set_header X-NginX-Proxy true;
      proxy_set_header Upgrade $http_upgrade;
      proxy_set_header Connection "upgrade";
      proxy_pass http://feedbase_server;
      proxy_hide_header X-Powered-By;
      proxy_redirect off;
    }

    # Serve static files with the following extensions from public folder
    location ~* ^.+\.(jpg|jpeg|gif|png|ico|css|zip|pdf|txt|js|html|htm)$ {
      # Adjust path to your setup
      root   /path/to/feedbase-version3/public;
    }
  }
```

8. Check if the configuration is valid:

   ```
   sudo nginx -t
   ```

9. If no errors occur, reload the nginx configuration

   ```
   sudo systemctl reload nginx
   ```

   The application should now be running through nginx.

References

[1] N. Foundation., “Node.js.” nodejs.org. [Online; accessed 08-June-2018].

[2] G. Inc., “Angular.js.” angularjs.org. [Online; accessed 08-June-2018].

[3] G. Inc., “Heatmap layer,” 2018. [Online; accessed 03-June-2018].

[4] G. Inc., “Map and tile coordinates,” 2018. [Online; accessed 03-June-2018].

[5] E. W. Weisstein, “Mercator projection.” From MathWorld–A Wolfram Web Resource. [Online; accessed 07-June-2018].

[6] stlwebdev, “Google map heatmap radius in meters.” Codepen. [Online; accessed 08-June-2018].

[7] M. de Berg, M. van Kreveld, M. Overmars, and O. C. Schwarzkopf, *Computational Geometry*, pp. 1–17. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000.

[8] M. Duckham, L. Kulik, M. Worboys, and A. Galton, “Efficient generation of simple polygons for characterizing the shape of a set of points in the plane,” *Pattern Recognition*, vol. 41, no. 10, pp. 3224 – 3236, 2008.

[9] T. P. D. Group, “Postgis st_concavehull,” 2018. [Online; accessed 15-March-2018].

[10] B. (https://stackoverflow.com/users/3877726/blindman67), “Sort points in counter clockwise order in javascript.” Stackoverflow. [Online; accessed 07-June-2018].