Bachelor Thesis

Managing and Querying Derived Nutrient Parameters in the Swiss Feed Database

Hannes Tresch
Sarnen, Switzerland
Matrikel-Nr. 08-715-534

supervised by
Prof. Dr. M. Böhlen and F. Cafagna

Department of Informatics,
University of Zurich

November 30, 2011

Abstract

The aim of the thesis has been to integrate the computation of derived nutrients into the Swiss animal feed database. Derived nutrients are parameters calculated by a formula that involves other nutrient parameters. The computation of these parameters replaces expensive chemical lab analyses. In order to get a meaningful temporal distribution of derived nutrients, an efficient method that supports time-varying regressions must be found.

For that, different SQL-queries, SQL-views and PL/pg SQL functions are implemented and presented in this document. The most efficient solution is then used for the implementation of an extension to the actual web application, so that all the functionalities that currently exist for non-derived nutrients are to be supported for derived nutrients as well.

Zusammenfassung

Das Ziel dieser Arbeit bestand darin, eine automatisierte Berechnung von abhängigen Nährwerten in die schweizerische Futtermitteldatenbank zu integrieren. Diese sogenannten 'abgeleiteten' Nährwerte (engl. *derived nutrients*) werden nicht durch chemische Analysen ermittelt, sondern sind abhängig von anderen Nährwerten und werden mittels Formeln berechnet. Um aussagekräftige Daten über die zeitliche Veränderung von abgeleiteten Nährwerten zu erhalten, musste eine Methode ausgearbeitet werden, die eine zeitbezogene Regressionsberechnung unterstützt.

Dazu werden in dieser Arbeit verschiedene Ansätze präsentiert, die mit Hilfe von SQL-queries, SQL-views und PL/pg SQL -functions umgesetzt wurden. Die effizienteste Implementierung wurde dann in die aktuelle Web Applikation integriert, sodass alle Funktionalitäten auch für abhängige Nährstoffe unterstützt sind.

Contents

List of Figures

List of Tables

Chapter 1

Introduction to the Swiss Feed Database

This thesis is part of the Swiss Feed Database project. The aim of the project is to produce a public service for companies, private farmers, and research institutions to get information about several nutrient parameters of specific feed types. This information can be used for an optimal and quality-conscious choice of feeds for a specific animal type. The database contains currently information for 155 nutrients and over 600 animal feed types. These data are collected through chemical analyses on feed sample measurements that are taken from all parts of Switzerland. All the information about the nutrients is stored in a database and can currently be accessed by a web application.[1][1]

At the moment, the University of Zurich collaborates with Agroscope (Bundesamt für Landwirtschaft) to design and implement new database techniques in order to improve the analysis of the feed data. In particular, an analysis of time-varying feed data for a desired period and specific biological or geographical parameters is required. Figure 1.1 shows the actual web application, Figure 1.2 is a new web application that is currently being developed at University of Zurich. In the new web application based on a new database design (Version 2.0),0 information about nutrients for specific geographical conditions and desired time periods can be retrieved and displayed in suitable form.

[1]The current web application can be accessed here: http://www.feed-alp.admin.ch/start.php

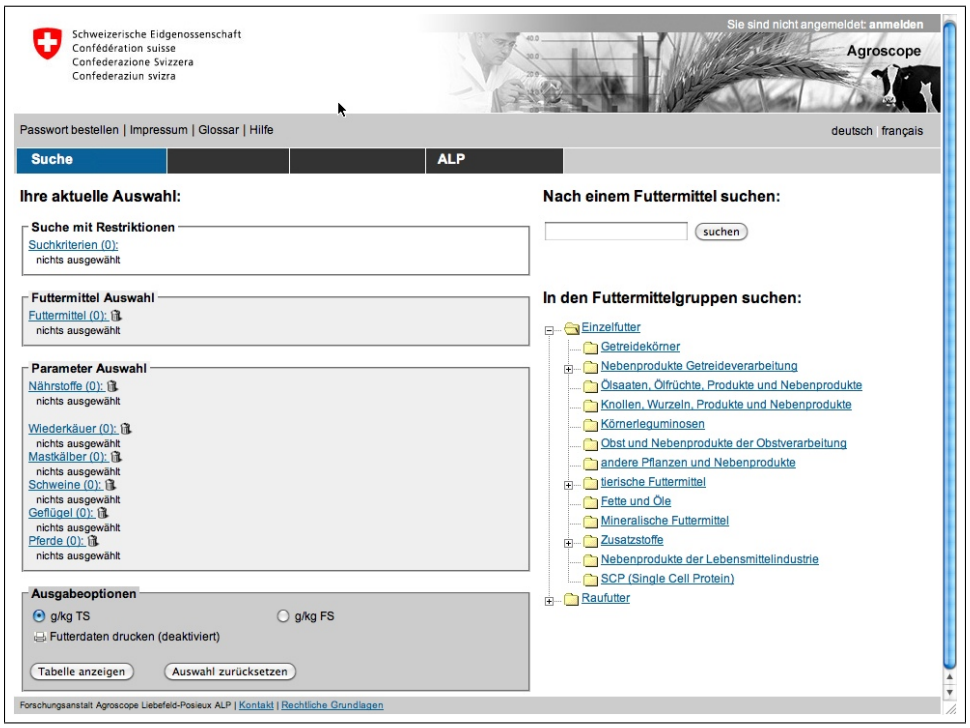
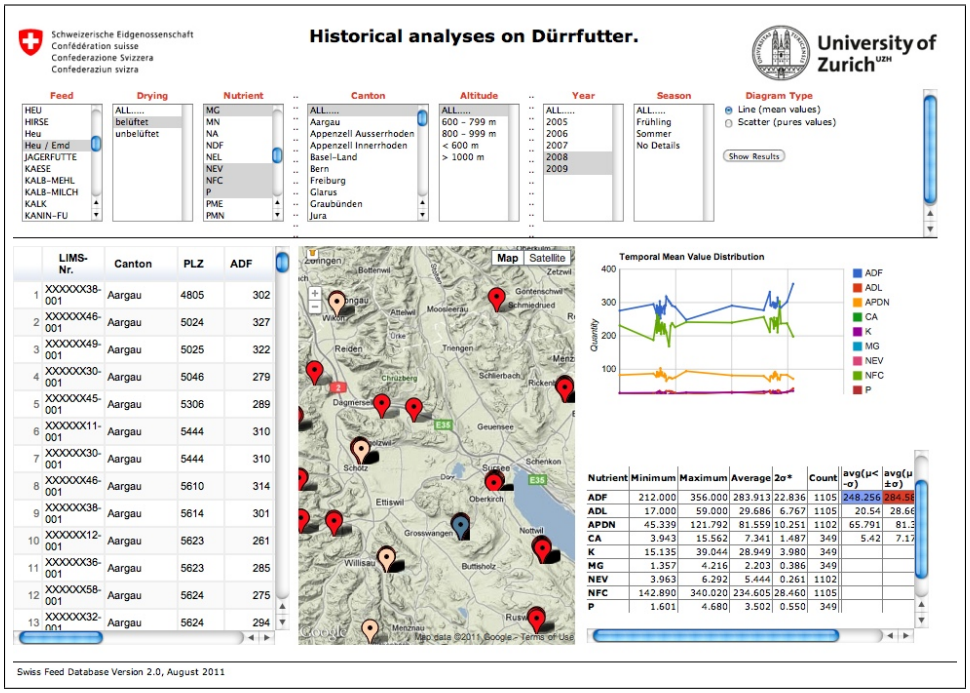


Figure 1.1: Web application, Version 1.0



	LIMS-Nr.	Canton	PLZ	ADF
1	XXXXXX38-001	Aargau	4805	302
2	XXXXXX46-001	Aargau	5024	327
3	XXXXXX49-001	Aargau	5025	322
4	XXXXXX30-001	Aargau	5046	279
5	XXXXXX45-001	Aargau	5306	289
6	XXXXXX11-001	Aargau	5444	310
7	XXXXXX30-001	Aargau	5444	310
8	XXXXXX46-001	Aargau	5610	314
9	XXXXXX38-001	Aargau	5614	301
10	XXXXXX12-001	Aargau	5623	261
11	XXXXXX36-001	Aargau	5623	285
12	XXXXXX58-001	Aargau	5624	275
13	XXXXXX32-001	Aargau	5624	294

Nutrient	Minimum	Maximum	Average	2σ*	Count	avg(μ<-σ)	avg(μ±σ)
ADF	212.000	356.000	283.913	22.836	1105	248.256	284.58
ADL	17.000	59.000	29.686	6.767	1105	20.54	28.66
APDN	45.339	121.792	81.559	10.251	1102	65.791	81.3
CA	3.943	15.562	7.341	1.487	349	5.42	7.17
K	15.135	39.044	28.949	3.980	349		
MG	1.357	4.216	2.203	0.386	349		
NEV	3.963	6.292	5.444	0.261	1102		
NFC	142.890	340.020	234.605	28.460	1105		
P	1.601	4.680	3.502	0.550	349		

Figure 1.2: Web application, Version 2.0

Chapter 2

Task definition and overview

The nutrient parameters of several different animal feeds are measured by chemical analyses. Apart from these measured data values, some nutrient parameters are computed by the help of a formula. This formula is an algebraic expression that involves a various number of nutrient components. The computation of such derived nutrients replaces expensive chemical lab analyses and results in approximate values.

Because it is hard to update all the derived nutrient values manually, a method that computes derived nutrients automatically is needed. In addition to that, the history of derived nutrient values should be stored in an appropriate way, so that based on the temporal distribution of these values, meaningful information can be extracted.

So, the aim of the thesis can be summarized as follows:

- Integrate the computation of derived nutrients into the Swiss feed database and
- implement an extension to the web application that supports time-varying regressions.

The thesis in hand is structured as follows: In a first step, the schemas that are used in this project are presented with some sample queries to get familiar with the database. After that, the derived nutrients are analyzed, classified and possible methods to compute them are presented.

Initially, it has been supposed to use views as a support to retrieve the values for derived nutrients. Because of performance problems, an alternative solution with a procedural language will be proposed.

Finally, some PL/pg SQL functions are defined that compute the values of derived nutrients in a temporal way. With the help of these functions, an extension to the web application (Version 2.0) is implemented, so that all functionalities of the web application will be supported for derived nutrients.

Chapter 3

Database schemas

3.1 Schema Version 1.0

The database schema of the Swiss feed database, Version 1.0, is shown in a simplified version in Figure 3.1. The illustration contains just the most important relations to give an overview on the database schema.

The most of the implementation approaches that are presented in chapter 5 are specified on the second version of the database. Nevertheless, the schema of the first version is presented here, because it is used for the first approach described in chapter 5.1.

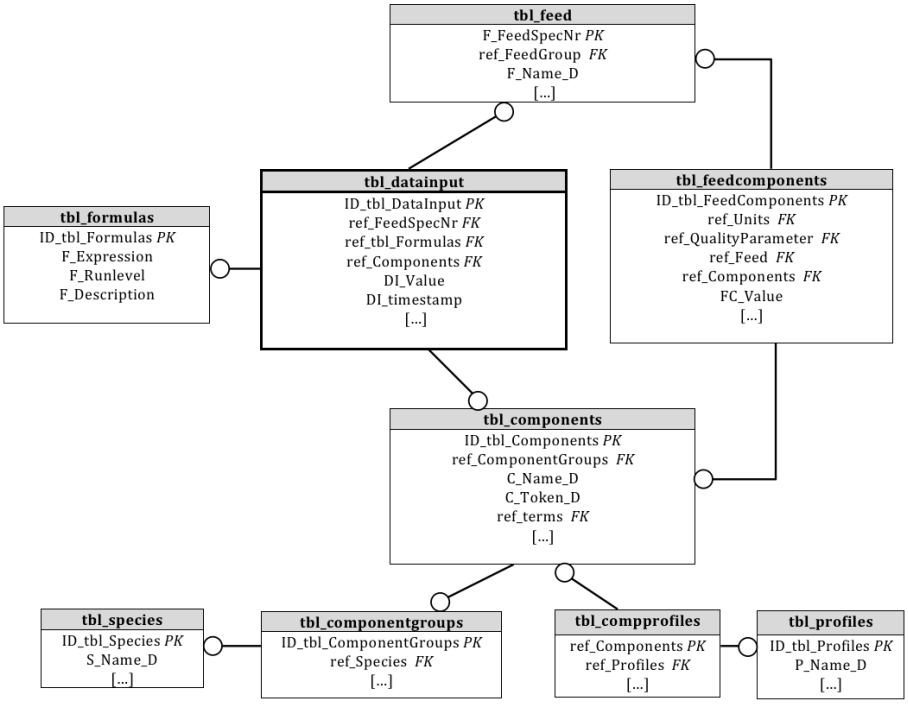


Figure 3.1: Simplified schema of the Swiss feed database, Version 1.0

In contrast to the second version, which is presented in chapter 3.2, not all the single measurement values with their corresponding temporal information are stored in the database. Instead of storing all the measurement values, just aggregated values are stored for nutrients on specified feeds. The most important tables from the simplified schema that is illustrated in Figure 3.1. are described below:

tbl_datainput: The table `tbl_datainput` in the middle of the illustration in Figure 3.1 contains the measured data values. The measurement value is stored in the attribute `DI_Value` together with the attribute `DI_timestamp` that indicates the measurement time. As a foreign key acts for instance the attribute `ref_FeedSpecNr` that references to a specific feed.

tbl_feed: This table contains all the feeds from which measurement samples are taken from. Each feed has a name and an ID (`F_FeedSpecNr`) that acts as primary key.

tbl_components: In the table called `tbl_components`, the nutrient names and the corresponding abbreviations are stored.

tbl_feedcomponents: This relation stores aggregated measurement values of different nutrient components that are measured in a various number of feeds.

tbl_formulas: The formulas of dervied nutrients are stored in the variable `F_Expression` of the table `tbl_formulas`. The primary key `ID_tbl_Formulas` contains the abbreviation of a derived nutrient. This abbreviation starts always with the number sign (`#`) to indicate that the nutrient is derived.

To familiarize with this database schema, two basic sql queries are defined below:

Retrieve the formulas of all derived nutrients with their related abbreviation

```
select distinct ref_tbl_formulas, f_expression
from tbl_datainput, tbl_components, tbl_formulas
where ref_tbl_formulas not like "%const%" and tbl_datainput.ref_components =
    id_tbl_components and tbl_formulas.id_tbl_formulas = ref_tbl_formulas
order by c_token_d;
```

Retrieve the value of the nutrient C14:0 that is measured for the feed with ID=62

```
select tbl_components.*, tbl_feedcomponents.*
from tbl_feedcomponents, tbl_components, tbl_units where (ref_unitgroups=1 or
    ref_unitgroups=0) and c_token_d like "C14:0"  and ref_units=id_tbl_units and
    ref_feed=62 and tbl_feedcomponents.ref_components=id_tbl_components
order by c_outputorder;
```

3.2 Schema Version 2.0

Figure 3.2 depicts the schema of the Swiss Feed Database (Version 2.0). The table that is called `fact_table` is the center of the schema. It contains all the information of a measurement that is taken from a specific sample. The `measure_pk` is the primary key of this table. Apart from the measurement value (`quantity`) and the sample identification number (`lims_number`) there are some other fields that act as foreign keys refering to other relations.

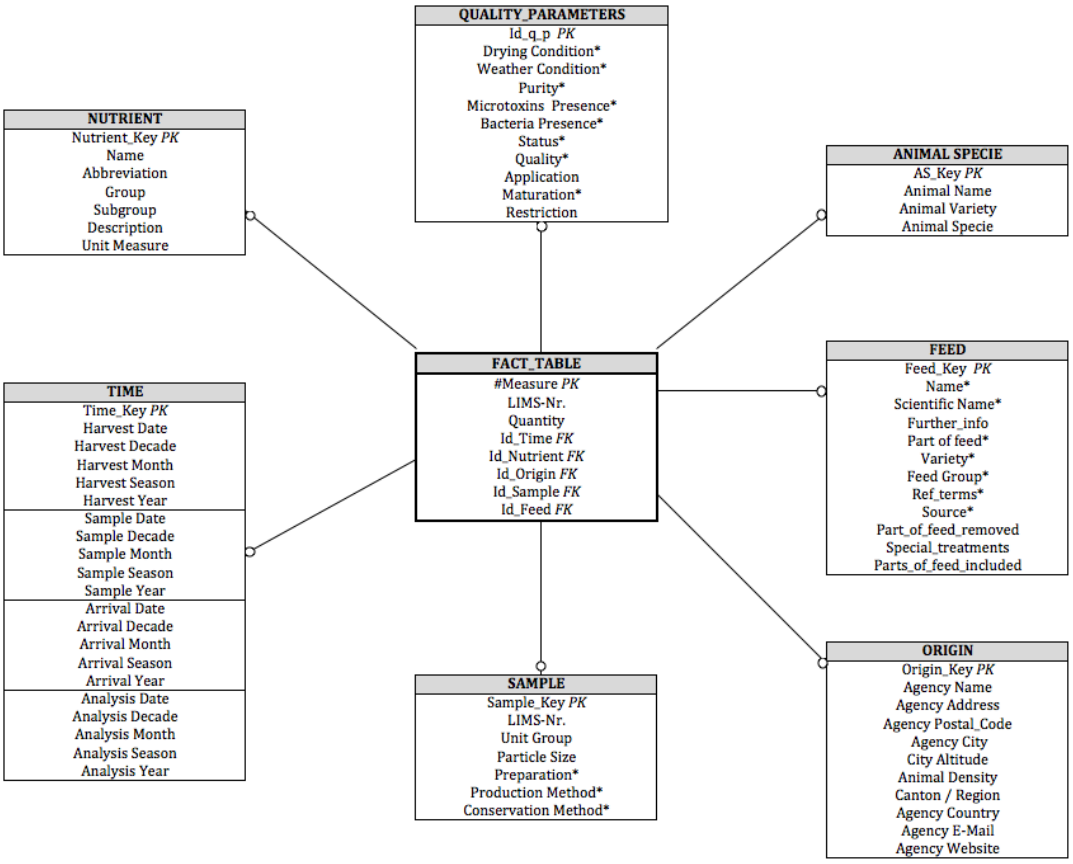


Figure 3.2: Schema of the Swiss feed database, Version 2.0

Because of the special representation of the table named time, this relation has to be mentioned in detail: As illustrated in Figure 3.2 , it can be stated four different time details in relation to a certain measurement. These are the `Harvest Date`, the `Sample Date`, the `Arrival Date` and the `Analysis Date`. This means, a measurement can be related to more than just one timestamp. The table `time` has an attribute named `moment` that identifies what kind of date the `time_key` is representing. So, the table `fact_table` can contain up to four tuples for the same measurement value, but with a different `time_key`. This ensures that all the different details according the measurement time are stored in an efficient way.

Chapter 4

Analyses and classification of derived nutrients

4.1 Classification of derived nutrients

The values of derived nutrient parameters are computed with algebraic expressions which involve a various number of nutrient parameters. These nutrient parameters can be

- values that are measured by chemical analyses, or
- derived nutrients that are calculated based on other derived nutrient parameters.

The latter implies that such regression can also be recursive. Table 4.1 shows some sample formulas of derived nutrients. The abbreviations of the nutrients in Table 4.1 are those of the first database version. The derived nutrients in row 2, 4 and 5 correspond to a formula that is recursive, whereas the derived components are in bold. In the square brackets after the abbreviations, the measuring unit is defined, followed by the drying reference.[1]

ref_tbl_formulas	f_expression
#Biotin[åµg_kg FS]	TS[g_kg] * Biotin[åµg_kg TS] / 1000
#BE_Maisganzpfl[MJ_kg TS]	0.0196 * **OS[g_kg TS]**
#OS[g_kg TS]	1000 - RA[g_kg TS]
#C20:4n-6[g_kg FS]	TS[g_kg] * **C20:4n-6[g_kg TS]** / 1000
#MPP_NEL[kg Milch_kg TS]	**NEL[MJ_kg TS]** / 3.14
#NEL[MJ_kg TS]	0.9752 * (0.463 + 0.24 * **q**) * UE[MJ_kg TS]

Table 4.1: Sample derived nutrients with their formulas

The principle of recursive formulas is illustrated by the simple formula of the nutrient named #BE_Maisganzpfl[MJ_kg TS] that is listed in the second row of the table above.

Start formula: 0.0196 * **OS[g_kg TS]**

↓

Expanded formula: 0.0196 * (1000 - RA[g_kg TS])

[1] TS = Trockenfutter, FS = Frischfutter

4.2 Analyses of nutrient abbreviations and formula expressions

The computation approaches that are presented in the chapters 5.4 and 5.5 are based on the principle of a formula evaluator. This means, that all the abbreviations that occur in a formula are extracted from this formula. For each abbreviation is then a corresponding measurement value assigned, such that the computation can be performed and a derived nutrient value results.

To extract all the involved abbreviations from the formula correctly, the structure of the abbreviations has to be analyzed to define in what way the formulas have to be written. For that, all the nutrient abbreviations of database schema 2.0 were analyzed and grouped in an appropriate way. In Table 4.2, these groups are listed with a corresponding example.

group	description	abbreviation_de	name_de
Group 1	just letters [A-Z]	GLY	Glycin
Group 2	[0-9] and [A-Z]	C3OH	Milchsäure
Group 3	'_'-symbol is involved	K_VA	Retinol (Vitamin A)
Group 4	'-' -symbol is involved	CU-PF	Kupfer Pet-food

Table 4.2: Classification of nutrient abbreviations

In most cases, the abbreviation names are only composed of letters and numbers. In some cases the hyphen- or the underscore-symbol is involved (group 3, resp. group 4). Specially if the abbreviation contains a hyphen-symobl, special consideration is required such that the hyphen is not recognized as a minus operator. For that, it is important to put a blank (␣) before or after each minus-operator that is involved in a formula. For all the other mathematical operators, it doesn't depend whether there is a blank between the operator and the abbreviation. The list below illustrates this rule and shows the kind of formulas that are supported by the PL/pg functions that are presented later in chapter 5.5.

1.	K_VA^2␣+␣(CU-PF␣-␣C10)␣*␣C3OH␣+␣GLY␣/␣2	✓
2.	K_VA^2+(CU-PF-C10)*C3OH+GLY/2	✗
3.	K_VA^2+(CU-PF␣-C10)*C3OH+GLY/2	✓

Table 4.3: Supported and not supported formula expressions

The second expression in this list is not valid, because the expression CU-PF-C10 is considered one nutrient. To split up the expression correctly in those base components, there must be an associated blank for each minus-operator. This is the case in example formula number 1 and and also in example formula number 3. However, the second formula is not supported by the algorithm approaches that are described in the chapters 5.5.

For all the formulas of derived nutrients, the most common mathematical operators as '+', '-', '*', '/' and '^' are used. All these are supported by postgreSQL whereby the computation of derived nutrients within pg SQL should be possible without any restrictions.[5]

Chapter 5

Design of Views, Queries and Algorithms

5.1 Materialized views

5.1.1 Introduction to views in SQL

In a first approach, it was supposed to use views for the computation of the derived nutrients. According to SQL terminology, a view can be defined as a virtual table that is derived from other tables. So, a view defines a function from a set of base stored tables to a derived table. Every time a view is referenced, this function is called and the virtual table is filled with tuples. A view can also be called a virtual table, because the tuples that are representing a view are not stored on a single table in the database. In contrast to that, in base stored tables, all the tuples of a relation are stored in the database.[3][4]

5.1.2 Views with up-to-date values

In a first phase, database schema Version 1.0 was used to create for each derived nutrient a view that is representing the up-to-date values for each feed type. So, it resulted in total 364 view definitions of different complexity. The query to create such a view is illustrated on the next page by the example of a basic formula.

The basic idea to get the up-to-date value of a derived nutrient is to use for the calculation the newest available data for each involved component of a given formula. The views are specified on the database schema version 1.0 and are composed of the following attributes:

1. The first attriubte is the ID of the feed type (`f_feedspecnr`).
2. The second attribute is the derived nutrient value (`value`) that is calculated in the view defintion as it is explained on the next page.
3. Additionally, the date that refers to a the specific derived nutrient value is specified in the third attribute (`timestamp`).

The listing on the next page shows a simple example how such a SQL view of a derived nutrient is created.

As an example, the derived nutrient with the abbreviation `#BE_Raufutter` is used:

Create a view that computes the up-to-date value of a derived nutrient

Derived nutrient:	#BE_Raufutter[MJ_kg TS]
Formula:	0.0188 * OS[g_kg TS] + 0.0078 * RP[g_kg TS]
View attributes:	f_feedspecnr | value | timestamp
Database schema:	Version 1.0

```
1  create view BE_Raufutter$MJ_kg$TS as
2  select f_feedspecnr,
3      (0.0188 *
4          (select cast(value as numeric)
5          from OS$g_kg$TS
6          where OS$g_kg$TS.f_feedspecnr=tbl_feed.f_feedspecnr and value is not null)
7
8      + 0.0078 *
9
10         (select cast(value as numeric)
11         from parameters, tbl_components, tbl_datainput
12         where ref_feedspecnr=id_feed and id_feed=f_feedspecnr and ref_components=
               id_tbl_components
13         and id_param=id_tbl_components and c_token_d='RP' order by tstamp desc limit
               1)
14     )
15
16     as value,
17
18     (select max(time_stamp)
19     from
20         ((select tstamp as time_stamp
21         from parameters, tbl_components, tbl_datainput
22         where id_param=id_tbl_components and id_feed=f_feedspecnr and f_feedspecnr=
               ref_feedspecnr and id_tbl_components=ref_components      and (c_token_d=
               'RP') and value is not null and ref_tbl_formulas='#constant')
23
24         union all
25
26         (select timestamp as time_stamp
27         from OS$g_kg$TS
28         where tbl_feed.f_feedspecnr=OS$g_kg$TS.f_feedspecnr)) as max_view
29     ) as timestamp
30
31 from tbl_feed order by f_feedspecnr;
```

As already mentioned, the calculation of the formula is performed with the newest available measurement values for each nutrient component. The formula that corresponds to the considered derived nutrient is composed of two numerical and two nutrient components. The nutrients that are involved are RP[g_kg TS] (Rohprotein) and OS[g_kg TS] (Organische Substanz):

- The newest available value for the second nutrient component named **RP[g_kg TS]** is retrieved in the subquery defined from line 10 to 13. This is done with the help of the LIMIT-clause. The join in the WHERE-clause (`id_feed=f_feedspecnr`) of the subquery guarantees, that for each feed just the corresponding data is used for the computation of a derived value.
- The other nutrient component that occurs in the formula is **OS[g_kg TS]** and can be identified as a derived component. This means that those values are stored in a view. The subquery from line 4 to 6 shows how the value for a specific feed is retrieved from a defined view.

In the select statement of the view definition above, the computation of the derived nutrient named #BE_Raufutter[MJ_kg TS] is done. For that, the selected values for RP and OS are used. This means that the formula (0.0188 * OS[g_kg TS] + 0.0078 * RP[g_kg TS]) is computed with the selected OS- resp. RP-value.

Finally, for each feed type, the ID of the feed (`f_feedspecnr`), the computed derived value and an associated timestamp is filled into the virtual table. The third attribute of the view named timestamp, represents the date of the derived nutrient value. This timestamp is the newest one of all the nutrient components of the formula. That can be done by a max-aggregation over all the involved components (line 18).

5.2 Standard SQL Implementation

5.2.1 Queries for time-varying regressions

The views that are presented in the previous chapter contain just the up-to-date values of a nutrient per each feed. In a next step, the views had to be adjusted, so that the history of a specific derived nutrient can be stored. An approach how time-varying regressions can be computed in a appropriate way is given in this sub-chapter.

The basic idea to get information of how the values of derived nutrients change over time is the following: For the calculation of a derived nutrient parameter at a specific time, we take measurement values of all involved components that have the same timestamp or are even from the same measurement sample. But in order to get a meaningful historical representation, we need more data values than just those that come from the same sample.

A possible solution to this problem is to take for all involved components the measurement values that are the closest to each other according to their timestamp.

The basic idea of this approach is explained by the following example: Assume that we have a derived nutrient with the abbreviation #RP_ALA that is calculated by the formula (RP * ALA) / 10. In this case, we create for RP and ALA a table that contains all the measurement values of the associated nutrient. The computing process of this example can be structured in two parts, whereas in each part the timestamps of a different component are in focus. These two parts are presented below:

1. Table 5.1 shows the first part, where the timestamps of RP are considered as fix. For each row in table RP, we search for a value in the other component where the timestamp is the closest to that of the RP value.

RP

id_feed	value	timestamp
1	3.1	2009-03-01
1	3.4	2011-09-01
1	5.0	2011-10-01
2	6.5	2000-04-03
2	7.1	2007-07-07

ALA

id_feed	value	timestamp
1	12	2003-02-01
1	10	2010-03-01
1	15	2011-10-01
2	14	2006-04-01
2	16	2007-07-04

Table 5.1: First combining pass considering the timestamps of table RP as fix

The first result value in Table 5.1 would be calculated by the expression (3.1*10)/10. The ALA-value from the second row is taken for the calculation, because the related

timestamp '2010-03-01' is the closest to the one in the first row of RP. This combinig and computing procedure is repeated for all RP-values.

2. In a second step, the same procedure is performed for the case where the timestamps of component ALA are considered as fix. This is illlustrated in Table 5.2.

RP

id_feed	value	timestamp
1	3.1	2009-03-01
1	3.4	2011-09-01
1	5.0	2011-10-01
2	6.5	2000-04-03
2	7.1	2007-07-07

ALA

id_feed	value	timestamp
1	12	2003-02-01
1	10	2010-03-01
1	15	2011-10-01
2	14	2006-04-01
2	16	2007-07-04

Table 5.2: Second combining pass considering the timestamps of table ALA as fix

The first five rows that are returned in this computing process are result values from the first computation pass where the timestamps of RP are considered as fix. The other five values come from the computation where the timestamps of ALA are in focus. So, it can be stated that a derived nurtient value is resulted for each timestamp that is present in one of those two tables. Table 5.3 shows all these result values for the fictive nutrient #RP_ALA that is calculated by the formula (RP * ALA) / 10. This result table is sorted by `id_feed` and by `timestamp` and contains all the timestamps. If there is a value for ALA that has the same timestamp as one of those in the RP value, this timestamp appears in the result table twice, although these two result values are probably the same. In the given result Table 5.3 this is the case in row 5 and 6.

id_feed	value	timestamp	*value calculated by*:
1	3.72	2003-02-01	*(12 * 3.1) / 10*
1	3.10	2009-03-01	*(3.1 * 10) / 10*
1	3.40	2010-03-01	*(10 * 3.4) / 10*
1	5.10	2011-09-01	*(3.4 * 15) / 10*
1	7.50	2011-10-01	*(5.0 * 15) / 10*
1	7.50	2011-10-01	*(15 * 5.0) / 10*
2	9.10	2000-04-03	*(6.5 * 14) / 10*
2	9.94	2006-04-01	*(14 * 7.1) / 10*
2	1.136	2007-07-04	*(16 * 7.1) / 10*
2	1.136	2007-07-07	*(7.1 * 16) / 10*

Table 5.3: Calculated derived nutrient values for #RP_ALA

How the described procedure of computing derived #RP_ALA-values can be transformed into a SQL Query is shown in the next listing. For that, two tables named RP and ALA are created which represent sample nutrients. The query that is applied to the real data set is shown afterwards.

Compute for each feed time-varying values of a derived nutrient

Derived Nutrient:	#RP_ALA
Formula:	(RP * ALA) / 10
Result attributes:	id_feed | value | timestamp

```
1  (
2    (select RP.id_feed, (RP.value * ALA.value) / 10 as value, RP.tstamp as timestamp
3    from RP, ALA
4    where RP.id_feed=ALA.id_feed and
5      ALA.tstamp
6        IN (
7          (select ALA2.tstamp
8          from ALA as ALA2
9          where RP.id_feed=ALA2.id_feed
10         order by abs(cast(ALA2.tstamp as date) - cast(RP.tstamp as date))
11         limit 1)
12       )
13       order by RP.id_feed)
14   union
15
16   (select ALA.id_feed as f, ALA.value * RP.value as v, ALA.tstamp as t
17   from RP, ALA
18   where RP.id_feed=ALA.id_feed and
19     RP.tstamp
20       IN (
21         (select RP2.tstamp
22         from RP as RP2
23         where  RP2.id_feed=ALA.id_feed
24         order by abs(cast(ALA.tstamp as date) - cast(RP2.tstamp as date))
25         limit 1)
26       )
27   )
28   order by id_feed, timestamp
29 );
```

The query above that retrieves time-varying values for a derived nutrient called #RP_ALA is described below:

1. From line 2 to 15, the first computation pass is done, where the timestamps of RP are considered as fix. This is previously visualized in the Table 5.1.

   (a) It retrieves the `id_feed`, the calculated `value`, and the `timestamp` of the RP measurement that is used for the computation.

   (b) In the subquery of the IN-clause (line 7 to 11), the timestamp of an ALA-measurement is retrieved that is the closest to the one of the considered RP-measurement. This is done by subtracting the timestamp values of the components that are considered. This guarantees that the ALA measurement is used, whose timestamp is the closest to that of the considered RP-measurement. Instead of doing the subtraction of timestamps, the postgres date-function `age(timestamp, timestamp)` could also be used to compare timestamps. [5]

2. From line 16 to 28, the second computation pass is done, where the timestamps of ALA are considered as fix. This is visualized in the Table 5.2 and works in the same way as previously described by the example of RP.

5.2.2 Queries specified on database schema Version 2.0

In a next step, the previously described query had to be adapted to the new database schema (Version 2.0). In the relation `fact_table` of the database, all the data that is taken from measurement samples is stored with the corresponding measurement time.

In the listing on the next page, the query to get the time-varying result values is defined on the database schema version 2.0. In the first part of the query, the timestamps of the first component are considered fixed. In the second query part, the timestamps of the second component are considered as fix. In order to get a final result table, all these query parts are put together with the UNION command.

Finally, the result table contains for each timestamp that appears in one of the involved nutrient tables a corresponding derived nutrient value.

The adapted query is illustrated by the example of the (fictive) derived nutrient with the formula TSO * ZUCK / 1000. The query in the next listing is just shown for the first computation part, where the timestamps of component TSO are considered as fix, but works accordingly for the other part.

Compute temporal derived nutrient values

Derived Nutrient (fictive):	#TSO_ZUCK
Formula:	TSO * ZUCK / 1000
Database schema:	Version 2.0
Result attributes:	id_feed | time_key | value

```
1  (select
2      TSO.id_feed, TSO.time_key, TSO.measurement * ZUCK.measurement/1000 as value
3  from
4      (select measurement, id_sample, id_feed, tday, id_nutrient, time_key
5      from d_nutrient, fact_table, d_time
6      where id_time=time_key and id_nutrient=nutrient_key and d_nutrient.
           abbreviation_de like 'TSO' and d_time.moment=2
7      )
8      as TSO,
9
10     (select measurement, id_sample, id_feed, tday, id_nutrient, time_key
11     from d_nutrient, fact_table, d_time
12     where id_time=time_key and id_nutrient=nutrient_key and d_nutrient.
           abbreviation_de like 'ZUCK' and d_time.moment=2
13     )
14     as ZUCK
15
16 where
17     TSO.id_feed=ZUCK.id_feed
18     and ZUCK.tday
19                 IN (
20                     (select ZUCK2.tday
21                     from
22                             (select id_feed, tday
23                             from fact_table, d_time
24                             where id_time=time_key and id_nutrient=ZUCK.id_Nutrient
                                   and d_time.moment=2 and
25                             ZUCK.id_feed=id_feed
26                             )
27                             as ZUCK2
28
29                     where ZUCK.id_feed=ZUCK2.id_feed
30                     order by abs(ZUCK2.tday - TSO.tday)
31                     limit 1
32                     )
33                 )
34 ) union [...]
```

The illustrated part of the query, where the timestamps of the nutrient named TSO are defined as fix, works as follows:

- In the SELECT-clause, the `time_key` is defined, together with the calculated result value named as `value`, and the `id_feed`.
- In the FROM-clause, the data for all the components that are involved in the derived nutrient are defined.
- With help of the SQL IN clause it is defined that the ZUCK measurement value with the closest timestamp to that of the TSO value is used for the calculation.

5.2.3 Explanation for performance problems

Unfortunately, the query outlined in the previous sub-chapter is not applicable to the given problem. The query takes several minutes to retrieve the computed result values for a derived nutrient that corresponds to a simple formula with only two involved nutrient components.[1]

The problem seems to be in the following join filter in the where-clause of the query:

"`and ZUCK.tday IN([subquery])`" (from line 19 in the listing on the previous page).

This where-condition that contains a subquery is used to guarantee that the ZUCK measurement with the smallest difference (in relation to their timestamps) to the considered TSO measurement is chosen for the computation. In order to calculate time-varying regressions of derived nutrients with in a standard SQL implementation, this join-filter with the subquery within the IN-clause is necessary.

With the help of the execution plan which can be generated in PostgreSQL with the `explain` command, the weak spots of a specified query can be located. This execution plan, also called query plan, includes the information about the estimated statement execution costs.[5]

An extract of the execution plan for the problematic query is shown below.

```
1  Hash Join  (cost=12898.47..281414500.24 rows=323245 width=40)
2  Hash Cond: (public.fact_table.id_feed = public.fact_table.id_feed)
3    Join Filter: (SubPlan 1)
4  ->  Nested Loop  (cost=0.00..48404.90 rows=5169 width=20)
5          ->  Nested Loop  (cost=0.00..21621.98 rows=6393 width=20)
6                [...]
7          ->  Index Scan using d_time_pkey on d_time  (cost=0.00..4.18 rows=1 width=8)
8                [...]
9    ->  Hash  (cost=12833.86..12833.86 rows=5169 width=24)
10         ->  Hash Join  (cost=386.70..12833.86 rows=5169 width=24)
11               Hash Cond: (public.fact_table.id_time = public.d_time.time_key)
12               ->  Nested Loop  (cost=122.04..12461.57 rows=6393 width=24)
13                   [...]
14               ->  Hash  (cost=200.23..200.23 rows=5155 width=8)
15       [...]
16   SubPlan 1
17     ->  Limit  (cost=435.17..435.17 rows=1 width=4)
18           ->  Sort  (cost=435.17..435.21 rows=18 width=4)
19                 Sort Key: (abs((public.d_time.tday - $0)))
20                 ->  Nested Loop  (cost=235.41..435.08 rows=18 width=4)
21                       ->  Bitmap Heap Scan on fact_table  (cost=235.41..305.88 rows=18
                             width=4)
22                             Recheck Cond: ((id_feed = $2) AND (id_nutrient = $1))
23                             ->  BitmapAnd  (cost=235.41..235.41 rows=18 width=0)
24                                   ->  Bitmap Index Scan on index_f_feed  (cost
                                         =0.00..114.71 rows=6162 width=0)
25                                         Index Cond: (id_feed = $2)
```

[1] Example components: ZUCK: ~ 9000 tuples; TSO: ~ 60'000 tuples

```
26                                    ->  Bitmap Index Scan on index_f_nut  (cost=0.00..120.44
                                            rows=6393 width=0)
27                                          Index Cond: (id_nutrient = $1)
28                              ->  Index Scan using d_time_pkey on d_time  (cost=0.00..7.16 rows=1
                                    width=8)
29                                    Index Cond: (public.d_time.time_key = public.fact_table.
                                          id_time)
30                                    Filter: (public.d_time.moment = 2)
```

As estimated, the main reason for the efficiency problem in the described query seems to be in the join filter of `SubPlan 1` (line 3 of the execution plan above). The `SubPlan 1` references to the part of the query where the data is filtered with the condition that the timestamp of the temporary table ZUCK must correspond with the timestamp that is retrieved from the subquery within the IN-clause (from line 22 in the listing on the previous page).

The total estimated execution cost (measured in disk page fetches) for this query can be read out from line number 1. The large part of this cost is caused by the mentioned join filter. This total estimated cost value can be compared with that of a query where we have just the join-condition on the two involved components (`TSO.id_feed=ZUCK.id_feed`).

query description	total estimated execution cost (in disk page fetches)
query from 5.2.2 **without** join filter condition "`ZUCK.tday IN([subquery])`"	~ 35'000
query from 5.2.2 **with** join filter condition "`ZUCK.tday IN([subquery])`"	~ 280'414'500

Table 5.4: Comparison of estimated execution costs

In the second case, where the restriction condition (from line 22 to 40 in the query on the previous page) is obmitted, the total cost amounts about 35'000 disk page fetches, which is only a fraction part of the cost that are measured for the complete query.

Because the mentioned join filter is essential to retrieve the results of the computation as time-varying regressions as it is proposed this approach is not applicable as a standard SQL Implementation.

In a next step it was proposed to use window functions to find a suitable solution for the presented problem. An approach to that is given in the next subchapter.

5.3 Window Functions

A window function in PostgreSQL can be defined as a function that performs a calculation across a set of tuples. This means, that data which are related somehow to a considered row can be retrieved from other rows. With the help of pg SQL Window Functions, the following two main purposes can be served:

- **cumulative computations:** Access to another row (for example the next or the previous row) and use those values for the calculation

- **partitioning aggregations:** Aggregate calculation over rows of a query result

To find a solution for the given problem of the calculation time-varying regressions, it makes sense to focus on window functions that pursues the purpose of cumulative computations. To perform cumulative computations, a row of a table has access to a set of rows and is able to use values of specific columns to make calculations. This means that for each row a calculated value can be returned. [5][6]

With the built-in window functions that are called `lag()` and `lead()`, values of the row below or the row above can be used for a calculation in a specific row. That means in fact, that is possible to make a calculation with parameter values that come from different rows. The returned result value can then be inserted in an additional column as illustrated in the example in Table 5.5.

id_feed	ZUCK_value	TSO_value	year	calc_lag	calc_lead
1900	4.1	-	1995	-	-
1900	-	5.2	1996	5.2 * 4.1	5.2 * 3.5
1900	3.5	-	1998	-	-
1900	-	6.0	1999	6.0 * 3.5	-
1900	-	7.0	2001	-	7.0 * 2.0
1900	2.0	-	2003	-	-

Table 5.5: Approach with window functions `lag()` and `lead()`

Unfortunately, I couldn't elaborate a working solution for the given problem using window functions. Nevertheless, the approach of using window functions for the computation of derived nutrients is presented in the following section by an example:

Assume that we have the formula ZUCK * TSO for a fictive derived nutrient and the Table 5.5 that contains some measurement values for the involved components. All these measurement values are ordered by time. The idea is now to add to each row two different result values that are calculated as follows with the help of the window functions `lag()` and `lead()`. The result value `calc_lag` is computed by multiplying the ZUCK-value of the current row with the TSO-value of the row above. The value of `calc_lead` is computed by using the TSO-value from the row below.

It is quite evident, that this solution is not working for the given problem. It would work only if there is alternately a ZUCK-measurement and a TSO-measurement. Because the table has to be ordered by timestamp, an alternate order is not possible.

So, the main problem can be explained by considering the following example: In the fifth row, the multiplication is only possible with the previous row. The result of the expression 6.0 * 3.5 is stored in `calc_lag`. For the same row, there is no result value in the column `calc_lead`, because it doesn't exist a ZUCK-value in the sixth row. In this case the `lag()`-function returns a null value.

To solve that problem, a partition should be created that contains just ZUCK-values and the fixed TSO-value. It is possible to create partitions over a specific nutrient as for example ZUCK, but then the TSO-value that is fixed is excluded from this partition and the calculation is not possible.

Other built-in window functions are proved to be not applicable for the given problem. So, an alternative solution with the help of an algorithm has to be found. In the next chapter an algorithm is presented, that calculates time-varying regressions in the way we want.

5.4 Implementation of Algorithm in Java

Because of the performance problems for the SQL-queries described in chapter 5.2 and the unsuccessful approach with window functions, it was necessary to find an alternative way for computing temporal regressions of derived nutrients. For this purpose, the SQL-approach presented in chapter 5.2, is transformed into an algorithm.

The basic idea of the algorithm is to loop over a sorted list of nutrient measurements and compare their timestamps to those of other component measurements. To perform this algorithm in an efficient way, two "pointers" keep track of the actual position in the component tables. The algorithm can basically be structured into two parts:

- **Combining process:** Search for each component measurement value a matching measurement value from the other components that are involved in the formula. This is done by the already explained "closest-timestamp"-principle.
- **Calculation process:** Calculates the derived nutrient value by replacing the nutrient abbreviation with a related value that is chosen in the combining process.

The algorithm of this combining and computing procedure is explained in this sub-chapter. In a first step, it is implemented as a java application. Later on, this algorithm is implemented in a PL/pgSQL function, so that it can be used for the integration of derived nutrients into the web application.

At the beginning of the algorithm, it is created an array for each component of the formula that contains all the measurement values of a specific nutrient component. All these arrays are sorted by timestamp and added to an arraylist. The basic idea of the algorithm is now to loop through these arrays and compare the timestamps in an appropriate way.

By the example of the formula (ZUCK+TSO)/100, the algorithm is explained below:

ZUCK

id	timestamp	value
1	1992-11-25	43.6
2	1992-11-25	44.9
3	1992-12-02	14.7
4	1992-12-16	39.6
5	1992-12-16	65.1
6	1992-12-16	27.9
7	1992-12-16	27.5
8	1993-02-18	31.3
9	1993-02-18	32.7
10	1993-03-04	14.3
...	...	...

diff1 *diff2*

0 <= *6*

1 > *0*

TSO

id	timestamp	value
..	...	...
7	1992-11-19	451.16
8	1992-11-25	445.58
9	1992-11-26	452.69
10	1992-11-26	453.62
11	1992-11-26	459.28
12	1992-11-26	453.5
13	1992-12-02	459.28
14	1992-12-02	453.65
15	1992-12-07	459.32
...	...	...

Table 5.6: Combining process I: Timestamps of nutrient ZUCK are considered as fix

1. **Combining process:**

   (a) In a first step, the timestamps of the first component are considered as fixed. This case is illustrated in Table 5.6. So we consider a ZUCK-value and search for that value a matching TSO-value whose timestamp is the closest to that of the ZUCK-measurement.

   (b) The search for a matching value is done with the help of a while-loop. This while-loop is illustrated with the arrows on the right hand side of the TSO-table in Table 5.6. In our example, the difference of the timestamp that correspond to the row where id=1 in component ZUCK and the timestamp that corresponds to the row where id=8 is 0. This difference is stored in a variable. Then the index of the second variable is increased by one and the difference is calculated again. There we have a difference of 1, which is greater than 0. That's why the index of the second component decreases again. Finally, the located measurement value (id=8) is used for the calculation.

   (c) Then the located position in the TSO-array (id=8) is stored in a variable.

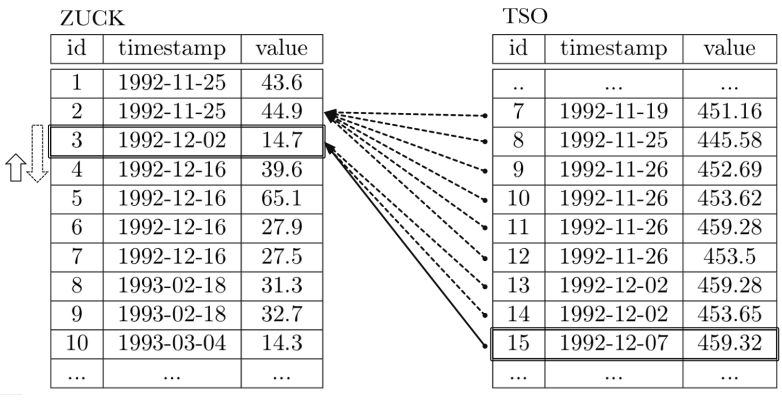
   (d) If there are more components, we loop through these lists as well and search for the value with the closest timestamp compared with that of the fixed component.

2. **Calculation process:**

   The calculation is done in a separate function, where for each abbreviations of the formula, the chosen values are assigned. This means that each abbreviation in the formula is replaced by a specific measurement value. In the java application, this is done with the help of a formula evaluator.

After the calculated result value for the first ZUCK-value is returned, the index of component ZUCK is increased and the same combining and computing procedure is performed for the next measurement in the first component (ZUCK).

In a next step the timestamps of the second component are considered as fix. This is illustrated in the Table 5.7 by the example of the TSO-measurement with id=15.



ZUCK

id	timestamp	value
1	1992-11-25	43.6
2	1992-11-25	44.9
3	1992-12-02	14.7
4	1992-12-16	39.6
5	1992-12-16	65.1
6	1992-12-16	27.9
7	1992-12-16	27.5
8	1993-02-18	31.3
9	1993-02-18	32.7
10	1993-03-04	14.3
...	...	...

TSO

id	timestamp	value
..	...	...
7	1992-11-19	451.16
8	1992-11-25	445.58
9	1992-11-26	452.69
10	1992-11-26	453.62
11	1992-11-26	459.28
12	1992-11-26	453.5
13	1992-12-02	459.28
14	1992-12-02	453.65
15	1992-12-07	459.32
...	...	...

Table 5.7: Combining process II: Timestamps of nutrient TSO are considered as fix

Finally, all the timestamps are fixed once, what means that we get for each measurement of an involved component a corresponding derived result value.

The description that follows explains the main steps of the implementation that is done in java:

1. As an input variable of the java application, we have a formula that corresponds to a specific derived nutrient and a feed number. First, all the component abbreviations must be extracted from the formula.. For that, different string-functions can be used. The goal is to create an array containing all the abbreviations that are in the formula. In the case of the example formula (ZUCK+TSO)/10, the abbreviations TSO and ZUCK are stored in an array called `abbr[]`.

2. In a second step, the java application retrieves from the database the required tuples. For each involved nutrient abbreviation where the `id_feed` corresponds to the one of the input parameter, all the tuples are retrieved.

3. Then, all the retrieved fields (`timestamp, value, id_feed`) are stored in a 2-dimensonal array. This array is then added to an arraylist that stores for each involved component in the formula a component-array with the data.

```
1  while (rs.next()) {
2          int timestamp = rs.getInt("timestamp");
3          double measurement = rs.getDouble("measurement");
4          int id= rs.getInt("id_feed");
5          comp[i][0] = timestamp;
6          comp[i][1] = measurement;
7          comp[i][2] = id;
8          i++;
9  }
10         list.add(comp);
```

4. As soon as for each component of the formula all the data is inserted, the described combining and computing algorithm starts. At first, the timestamps of the first component, in our example those of nutrient ZUCK, are fixed. Then, we search for all the other components a measurement value, whose timestamp is the closest to that of the ZUCK-measurement. In a first FOR-clause it is iterated through all the values of the first component (ZUCK). Then the difference between the timestamp of the second component and the first component is calculated and temporarily stored in a variable.

5. In a while-clause, the index of the second component (TSO) increases every time by one until the value of the timestamp-difference becomes bigger than the one of the stored variable. If that is the case, the index decreases by one and then the measurement value with the current is used for the calculation.

The listing below illustrates an extract of a sample result output. Line 7 shows the result attributes (timestamp: 20110318, calculated value: 40.88214, id_feed: 1900). The result value is calculated by the expression (66.52+342.3)/100.

```
1  ZUCK value: 66.5214
2  ZUCK timestamp 20090730
3
4  TSO value: 342.3
5  TSO timestamp: 20110318
6
7  Component 1 (TSO) is fixed:  result = 20110318, 40.88214, 1900
```

5.5 PL/pg SQL Functions

5.5.1 Introduction to PL/pg SQL Functions

In a next step it was the task to transform the algorithm that is implemented in java in a suitable form to a PL/pg SQL function. PL/pgSQL is a procedural language for the PostgreSQL database system. With the help of this language functions can be created and complex computations can be performed. The language supports common features in programming languages like variables, if-clauses or looping-clauses.[5]

5.5.2 Implementation approach using a 3-dimensional-Array

As mentioned in the previous chapter where an approach of a java implementation is shown, the idea is to store all the measurement values of the involved components in memory. This is done with the help of a 3-dimensional array named `allcomp[n][i][j]`, that is filled with all the measurement values that are used for the calculation of a specific derived nutrient. The meaning of the indices in each dimension in the allcomp-array is listed below:

n: indicates the nutrient component
(n=1 corresponds to the first involved nutrient in the formula)

i: indicates the current row

j: indicates whether it is a timestamp (j=1) or a measurement value (j=2)

Table 5.8 shows the first few data values of the `allcomp[][][]`-array for the fictive formula (ZUCK + TSO + ETOH) / 1000.

allcomp[1][i][j]		allcomp[2][i][j]		allcomp[3][i][j]	
timestamp	value	timestamp	value	timestamp	value
1992-11-25	43.6	1992-11-19	432	1992-11-25	1.3137
1992-11-25	44.9	1992-11-19	431	1992-11-26	0.8558
1992-12-02	14.7	1992-11-19	447	1992-12-02	0.9834
1992-12-16	39.6	1992-11-19	440	1992-12-10	0.8169
1992-12-16	65.1	1992-11-19	443.96	1992-12-16	1.1169
1992-12-16	27.9	1992-11-19	451.16	1992-12-16	1.1212
1992-12-16	27.5	1992-11-25	445.58	1992-12-16	0.6065
1993-02-18	31.3	1992-11-25	452.69	1993-02-18	1.5587
1993-02-18	32.7	1992-11-26	453.61	1993-03-02	2.1628
1993-03-04	14.3	1992-11-26	459.28	1993-03-02	0.9386
[...]	[...]	[...]	[...]	[...]	[...]

Table 5.8: Illustration of 3-dimensional-array `allcomp[][][]`

Compute temporal derived nutrient values

Derived Nutrient (fictive):	#ZUCK_TSO_ETOH
Formula:	(ZUCK + TSO + ETOH) / 1000
Database schema:	Version 2.0

The algorithm works in a similar way as it is already explained in the previous chapter by the example of the java application. The main steps of the implemented PL/pg SQL function can be summarized as follows:

Step 1: Expand the formula if it contains derived components (indicated by a #-sign)

Step 2: Extract from the formula all involved abbreviations and store them in an array.

Step 3: Fill a 3-dimensional `allcomp[][][]-array` with data from the database.

Step 4: Perform combining procedure:

allcomp[1][i][j]		allcomp[2][i][j]		allcomp[3][i][j]	
timestamp	value	timestamp	value	timestamp	value
1992-11-25	43.6	1992-11-19	432	1992-11-25	1.3137
1992-11-25	44.9	1992-11-19	431	1992-11-26	0.8558
1992-12-02	14.7	1992-11-19	447	1992-12-02	0.9834
1992-12-16	39.6	1992-11-19	440	1992-12-10	0.8169
1992-12-16	65.1	1992-11-19	443.96	1992-12-16	1.1169
1992-12-16	27.9	1992-11-19	451.16	1992-12-16	1.1212
1992-12-16	27.5	1992-11-25	445.58	1992-12-16	0.6065
1993-02-18	31.3	1992-11-25	452.69	1993-02-18	1.5587
1993-02-18	32.7	1992-11-26	453.61	1993-03-02	2.1628
1993-03-04	14.3	1992-11-26	459.28	1993-03-02	0.9386
[...]	[...]	[...]	[...]	[...]	[...]

Table 5.9: Combining process in the 3-dimensional array

Loop through all the measurement values of the first component and search for all other involved components the value with the timestamp that is the closest to the one of the first component. The index position of a chosen measurement value is stored in `jPos[n]`. The variable `n` stands for the nutrient index number and corresponds with those of the array `abbr[]`, that contains all the involved abbreviations.

An extract of the implementation of the described combining algorithm in PL/pg SQL is given on the next page. After that, the computing procedure is explained in step 5-7.

Combining procedure implemented in PL/pg SQL:

```
1  for x in 1 .. array_upper(allcomp, 2) LOOP
2      jPos[c1] := x;
3
4      if (allcomp[c1][jPos[c1]][1] is not null and allcomp[c1][jPos[c1]][1]!= 0) then
5          for c2 in 1 .. (array_upper(allcomp,1)-1) LOOP
6
7              if c1!=c2 then
8              actdiff:=abs(allcomp[c1][jPos[c1]][1] - allcomp[c2][jPos[c2]][1]);
9              diff := actdiff;
10
11                     while actdiff <= diff and jPos[c2]< array_length(allcomp, 2) and
                            allcomp[c2][JPos[c2]+1][1] is not null and allcomp[c2][JPos
                           [c2]+1][1] != 0 LOOP
12                         jPos[c2]:= jPos[c2] + 1 ;
13                         actdiff:= abs(allcomp[c1][jPos[c1]][1] - allcomp[c2][jPos[c2
                               ]][1]);
14
15                         if actdiff > diff then
16                             jPos[c2] := jPos[c2] - 1;
17
18                         else
19                             diff := actdiff;
20                         end if;
21                     end loop;
22             end if;
23         end loop;
24 [...]
```

Step 5: Replace all the abbreviations with the corresponding measurement values that are resulted from the combining process.

Step 6: Compute the result of the formula (formula contains now only numerical values because of the replacement that is performed in step 5).

Step 7: Return the computed value and the timestamp that was considered as fix.

The use of a multidimensional array for the given problem is probably not the most efficent solution. Especially in PL/pg SQL, the initalization of bigger arrays is not really cost-efficient in relation to the execution time of the function. [5]

Because of that, it was necessary to find other possible solutions with PL/pg SQL. These approaches are presented in the next subchapters.

5.5.3 Implementation approach using a string array

The idea of the approach that is presented in this subchapter is to use an one-dimensional array instead of a multidimensional array. For that, all the needed information of a measurement have to be stored in a string-variable. As a delimiter, the '§'-symbol is used in this example. In a seperate function, the desired values are extracted from the string variable and used for the computation of the result values. The procedure of the algorithm works in a similiar way as it is described in the previous chapter.

Compute temporal derived nutrient values	
Derived Nutrient (fictive):	#ZUCK_TSO_ETOH
Formula:	(ZUCK + TSO + ETOH) / 1000
Database schema:	Version 2.0

allcomp[]	
'§1§1992-11-25§43.6§'	measurement data for nutrient ZUCK
'§1§1992-11-25§44.9§'	
'§1§1992-11-02§14.7§'	
'§1§1992-12-16§39.6§'	
[...]	
'§2§1992-11-25§445.58§'	measurement data for nutrient TSO
'§2§1992-11-25§452.69§'	
'§2§1992-11-26§453.61§'	
'§2§1992-11-26§459.28§'	
[...]	
'§3§1992-11-25§1.3137§'	measurement data for nutrient ETOH
'§3§1992-11-26§0.8558§'	
'§3§1992-12-02§0.9834§'	
'§3§1992-12-10§0.8169§'	
'§3§1992-12-16§1.1169§'	
[...]	

Table 5.10: Illustration of string array allcomp[]

5.5.4 Implementation approach using cursor variables

Instead of storing the measurement information that is used for the calculation in memory, it is also possible to define cursor variables to iterate through a data set. In PostgreSQL, cursors can be defined as read-only pointers to a specified result set of a SQL-Query.[7] The use of a cursor is similar to that of a FOR-IN-SELECT-loop, but with the following difference: With paramterized cursors, it is possible to change the result set of a query. This can be done by setting formal parameters in the defintion of the cursors.[5][2]

The approach using cursors is explained by the example of the following formula: (ZUCK + TSO + ETOH) / 1000. Independent of the number of involved parameters, two cursor variables are defined in this implementation approach. In a first phase, the cursors are defined on the result set of the query that retrieves the ZUCK, respectively the TSO measurements. Then the cursors are defined on the result set of ZUCK and ETOH. These two phases are depicted in figure 5.8 and 5.9 for the first three ZUCK-measurements.

ZUCK

timestamp	value
1992-11-25	43.6
1992-11-25	44.9
1992-12-02	14.7
1992-12-16	39.6
1992-12-16	65.1
1992-12-16	27.9
1992-12-16	27.5
1993-02-18	31.3
1993-02-18	32.7
1993-03-04	14.3
[...]	[...]

TSO

timestamp	value
[...]	[...]
1992-11-19	451.16
1992-11-25	445.58
1992-11-26	452.69
1992-11-26	453.62
1992-11-26	459.28
1992-11-26	453.5
1992-12-02	459.28
1992-12-02	453.65
1992-12-07	459.32
[...]	[...]

ETOH

timestamp	value
1992-11-25	1.3137
1992-11-26	0.8558
1992-12-02	0.9834
1992-12-10	0.8169
1992-12-16	1.1169
1992-12-16	1.1212
1992-12-16	0.6065
1993-02-18	1.5587
1993-03-02	2.1628
1993-03-02	0.9386
[...]	[...]

Table 5.11: Combining pass with two cursors (on the result set of ZUCK, resp. TSO)

ZUCK

timestamp	value
1992-11-25	43.6
1992-11-25	44.9
1992-12-02	14.7
1992-12-16	39.6
1992-12-16	65.1
1992-12-16	27.9
1992-12-16	27.5
1993-02-18	31.3
1993-02-18	32.7
1993-03-04	14.3
[...]	[...]

TSO

timestamp	value
[...]	[...]
1992-11-19	451.16
1992-11-25	445.58
1992-11-26	452.69
1992-11-26	453.62
1992-11-26	459.28
1992-11-26	453.5
1992-12-02	459.28
1992-12-02	453.65
1992-12-07	459.32
[...]	[...]

ETOH

timestamp	value
1992-11-25	1.3137
1992-11-26	0.8558
1992-12-02	0.9834
1992-12-10	0.8169
1992-12-16	1.1169
1992-12-16	1.1212
1992-12-16	0.6065
1993-02-18	1.5587
1993-03-02	2.1628
1993-03-02	0.9386
[...]	[...]

Table 5.12: Combining pass with two cursors (on the result set of ZUCK, resp. ETOH)

The main steps of using cursors are listed below. Then the algorithm is explained by using the mentioned example formula that involves the nutrients ZUCK, TSO and ETOH.

1. **Declare a cursor:** Define a result set of a specific query
2. **Open a cursor:** Before a cursor can be used it has to be opened
3. **Fetch from a cursor**: Retrieve one row at a time from the result set
4. **Close a cursor**: Close a cursor after its use to release the context area

Compute temporal derived nutrient values with help of cursors

Derived Nutrient (fictive):	#ZUCK_TSO_ETOH
Formula:	(ZUCK + TSO + ETOH) / 1000
Database schema:	Version 2.0

Step 1: In a first step, two cursor variables are defined as `curs1` and `curs2` in the DECLARE statement of the PL/pg SQL function.

Step 2: In the BEGIN statement, the cursors are opened for a specified result set. Variable `curs1` is specified for a query that retrieves the data for the first component (ZUCK), whereas `curs2` loops on the data of the second component (TSO). With the help of the abbreviation array `abbr[]`, that contains all the involved abbreviations, this procedure can be done in a dynamic way for all the involved components.

```
1  for c1 in array_lower(abbr, 1) .. array_upper(abbr, 1) loop
2
3      for c2 IN array_lower(abbr, 1) .. array_upper(abbr, 1) loop
4
5      if c1!=c2 then
6
7      OPEN curs1 FOR (select (extract(Day from tday))+(extract(Month from
           tday)*100)+(extract(year from tday)*10000) as ts,
8          measurement
9          from d_nutrient, fact_table, d_time
10         where id_time=time_key and id_nutrient=nutrient_key and
               d_nutrient.abbreviation_de like abbr[c1] and d_time.moment=3
               and id_feed=feed order by id_feed, tday, ft_key);
11
12     OPEN curs2 FOR (select (extract(Day from tday))+(extract(Month from
           tday)*100)+(extract(year from tday)*10000) as ts,
13         measurement
14         from d_nutrient, fact_table, d_time
15         where id_time=time_key and id_nutrient=nutrient_key and
               d_nutrient.abbreviation_de like abbr[c2] and d_time.moment=3
               and id_feed=feed order by id_feed, tday, ft_key);
```

Step 3: In a next step, the data is fetched from the cursor variable into a record variable. Now, the fetched timestamps can be compared in the same way as it is done in the previous implementation approach: The `fetch next` statement is executed in a while loop until the interval between the timestamp of the first component and the timestamp of the other component is increasing. In case that the difference becomes bigger, the previous measurement value is fetched using the `fetch prior` statement. This value is finally used for the calculation.

Step 4: As soon as for each fetched value from `curs1` a value from `curs2` is fetched, the cursor variable `curs2` is closed. Cursor `curs2` is then opened again, but on a different result set. In this case, it is opened on the result set of the query that retrieves the data of the third component (ETOH).

Finally, all the formulas, whose abbreviations are replaced by the elaborated measurement values, are computed. It results the computed value and the timestamp of the measurement that was considered as fix.

Instead of using just two cursor variables as described above, it would be more efficient to declare for each involved nutrient component a separate cursor. How this procedure would work for the mentioned formula is illustrated in Table 5.13.

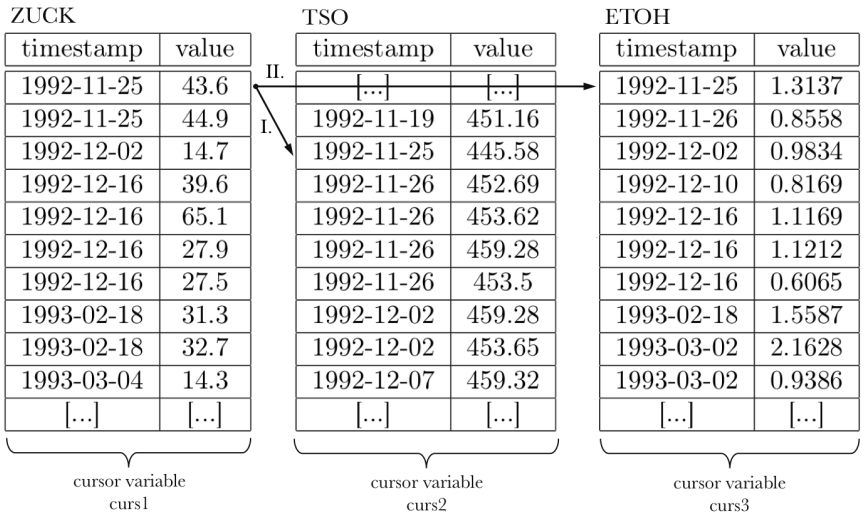


Table 5.13: Combing pass with ZUCK, TSO and ETOH using cursors

In a first step (I) the cursor that is declared on the result set of the ZUCK-query fetches the first value and searches a matching TSO-Value with help of the second cursor (`curs2`). Instead of fetching the next ZUCK value, a matching value from table ETOH is searched in a second step (II). In this case, the formula could be calculated directly with help of the chosen measurement values, before the next row of the ZUCK component is fetched.

This alternative solution seems to be more efficient, because the iteration on the timestamps of a specific component has to be performed just once. In the other case, where we declared just two cursor variables, it has to be iterated multiple times depending on the number of involved components.

An implementation where we assign to each nutrient component a corresponding cursor variable is problematic: To guarantee that the algorithm works for all the formulas with a various number of components, it is necessary to declare a number of cursor that is not predefined. The idea was to create an array that stores all the declared cursor variables. But this couldn't be realized, because no container could be figured out that supports storing cursor variables. That's the reason why this version hasn't been suitably implemented yet.

5.5.5 Performance comparison of implementation approaches

Table 5.14 shows a comparison of the presented PL/pg SQL functions according to their execution time. All the PL/pg SQL functions are tested for some sample formulas. In this performance comparison the functions are executed for the feed with `id_feed=1900` and the formulas that are listed in Table 5.14.

It must be mentioned, that the algorithm that is implemented in an analog way in java was more efficient according to the execution time. The reason for the difference between the function that is implemented in java and the function that is implemented in PL/pg SQL, can be explained with the fact, that java is running outside the RDBMS[2]. So, the processor resources are greater.

However, for the implementation of an extension to the web application, the use of a PL/pg SQL function is applicable. So the focus is set on the three different implementation versions of PL/pg SQL functions that were presented in chapter 5.5.2, 5.5.3 and 5.5.4.

sample formula	approach 1: float-[][][]	approach 2: string-[]	approach 3: cursors
(ZUCK + TSO)/100	4,7 sec	22,5 sec	8,6 sec
(ETOH+ZUCK+TSO+ #ADF_Gerste)/100	15,6 sec	72 sec	27 sec

Table 5.14: Performance comparison of PL/pg SQL functions

Table 5.14 shows the execution time for each approach by two different example formulas[3]. The reasons for the execution time of the respective approach are stated below:

- In the second implementation, the idea was to use a **1-dimensional string-array** to store the relevant measurement data. In such a case, it can be avoided to initialize a large 3D-array as it is used in the first approach. The string contains the id_feed, the timestamp and the measurement value and looks like that: "§1§19980402§342.1§".

  The long execution time for this approach is referable to the cost-intensive extraction of substrings to get the desired timestamp resp. the desired measurement value from the string variable.

- In the third approach, the algorithm is implemented by using **two cursor variables.** The advantage here is that the used measurement data has not to be stored in memory as it is the case in approach 1 and 2. Instead of that, cursors are declared on the result set of a specified query.

  A disadvantage of this implementation approach with 2 cursor variables is the fact that the loop over the result set has to be performed more than once for a component where the timestamps are considered as fix. For that reason, this solution is especially not applicable for formulas that contain a large number of nutrient components.

- So it can be said, that the approach that uses a **3-dimensional array** is the most applicable one for the given problem. But also with this solution, it can take several seconds to retrieve the computed derived values. Altough, this solution was integrated into the web application, so that all the functionalities are supported for derived nutrients. The main tasks for this integration are described in the next chapter.

[2]Relational Database Management System
[3]Number of involved tuples in the formula: 1st formula: 8983; 2nd formula: 12928

Chapter 6

Implementation of an extension to the Swiss Feed Database

6.1 Introduction to Swiss Feed Database web application Version 2.0

The web application of the Swiss Feed Database Version 2.0 has been developed at University of Zurich. With this application, information about specific nutrients can be retrieved and is visualized in suitable form. The layout of the current webpage consists of two main parts, which are presented in this sub-chapter.

- *selection part:* In this part, the user can define some restriction conditions in addition to the specific nutrients that the result should be computed for. In the first select field, a feed or a number of feeds can be selected for the calculation of the results. Other restrictions are way of drying, measurement time, and geographical conditions like 'canton' or 'altitude'.

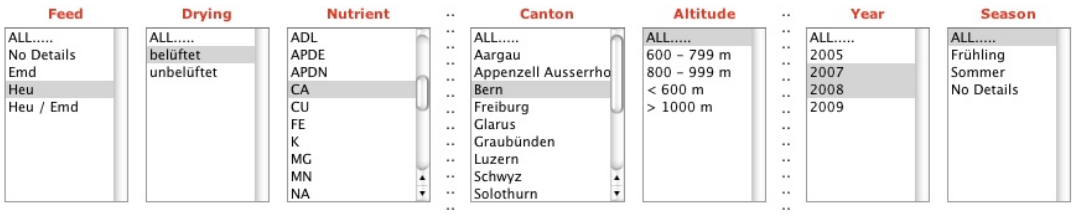


Figure 6.1: Selection part of the web application

- *result part:* The second part of the webpage is the result part, which appears as soon as the user clicks on the 'show results'-button. It is divided into three columns that contain the following information:

1. The first column is the **sample enlistment** that shows all the measurement values for the selected nutrients.

	LIMS-Nr.	Canton	PLZ	CA	CU	FE
1	XXXXXX88-001	Aargau	5054			
2	XXXXXX70-001	Aargau	5647			
3	XXXXXX70-002	Aargau	5647			
4	XXXXXX84-014	Aargau	5722	5.605	7.961	282.443
5	XXXXXX84-015	Aargau	5722	6.488	7.816	365.096
6	XXXXXX63-006	Aargau	5722			
7	XXXXXX63-011	Aargau	5722			
8	XXXXXX06-001	Basel-Land	4438			
9	XXXXXX25-001	Basel-Land	4494	10.639	7.584	510.293
10	XXXXXX31-008	Bern	2608	11.07	7.968	345.989
11	XXXXXX17-001	Bern	2733	6.505		
12	XXXXXX31-001	Bern	2733	9.387		
13	XXXXXX31-002	Bern	2733	5.215		

Figure 6.2: Sample enlistment

2. On the map in the middle of the result part, the location from which the measurement samples were taken are marked with pins.

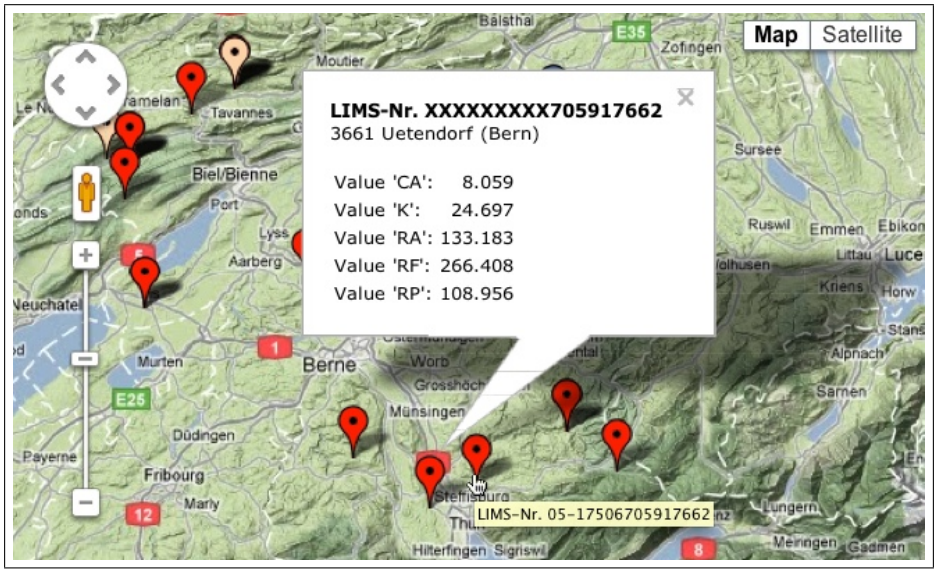


Figure 6.3: Map with marked measurement locations

3. On the right side of the web page is a **chart** that displays the temporal value distribution of the selected nutrients. The diagram can either be displayed as a line-diagram that takes the mean values for calculation of the graph, or as a scatter-diagram, in which each measurement value is displayed as a dot. Figure 6.4 shows for both diagram type an example.

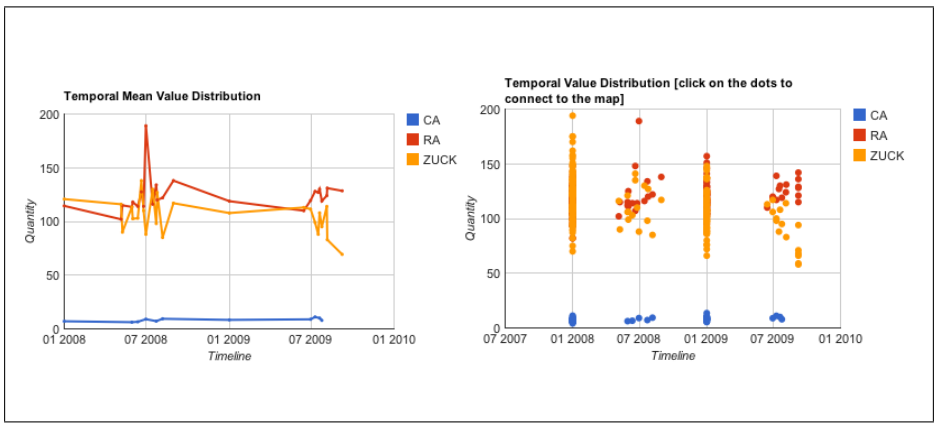
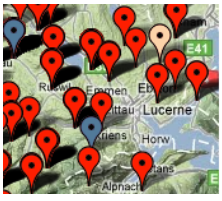


Figure 6.4: Line-diagram and Scatter-diagram

4. The HTML-table that is represented in Figure 6.5 gives **statistical information** for each selected nutrient. These are aggregate values like the maximum-, minimum- or average-value for each selected nutrient. The aggregate values are always computed with all the measurement values that satisfy the specified conditions. In order to detect outliers, the values in the last three columns of the aggregation table in Figure 6.5 are calculated. Depending on the measured data value in specific sample, the location is marked in an other color. The blue pins on the map identifies extremely small measurment values, wheras the yellow pins visualize the locations where extremly high values are for a specified nutrient parameter.



Nutrient	Minimum	Maximum	Average	2σ*	Count	avg(μ< -σ)	avg(μ±σ)	avg(μ> +σ)
CA	4.102	14.080	8.041	1.837	127	5.542	7.989	11.213
RA	81.072	341.714	123.003	26.479	465	91.021	119.335	191.528
SF	18.820	40.820	29.184	3.164	216	24.47	29.178	34.231
ZN	20.909	41.429	30.756	4.853	28	22.271	30.157	38.242
ZUCK	58.000	194.000	113.037	21.514	215	80.467	111.497	146.556

Figure 6.5: Aggregation table with statistical information

6.2 Overview of tasks for the integration of derived nutrients into the web application

For the web application to run also for derived nutrients, some adjustments had to be made. The four main implementation tasks are presented in this chapter:

1. Insertion of derived nutrients into the nutrient select field [Task 1]

In a first step, the functionality of automatically updating the select fields has to be adjusted. Depending on the selected options in the fields `feed` and `drying` the nutrient field has to be updated. The nutrient select field should list just options for which data exists. If there is no data, the nutrient abbreviation should not appear. This means for derived nutrients that for each nutrient component of the formula a checking has to be made whether some data exist.

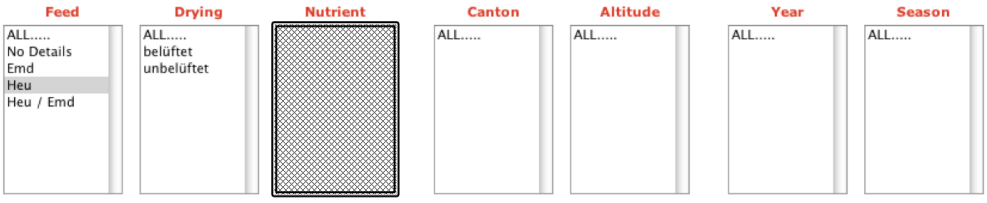


Figure 6.6: Visualization of task 1 concerning the integration of derived nutrients

2. Update select fields depending on the selected nutrients [Task 2]

The select field that follows the nutrient select field must be updated in the same way. So, the options of the fields `canton`, `altitude`, `year` and `season` have to be listed depending on the previously made selections. This means that for each component of a derived nutrient it has to be checked if there is data available or not.

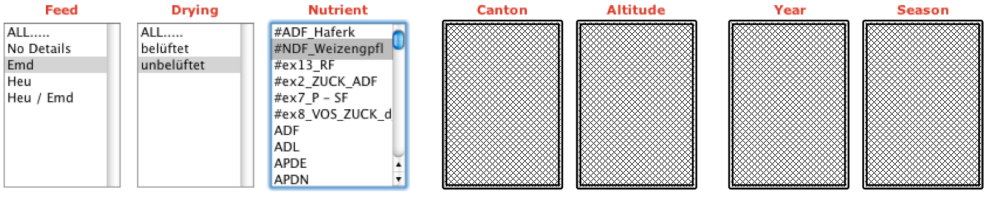


Figure 6.7: Visualization of task 2 concerning the integration of derived nutrients

3. Compute time-varying regressions for derived nutrients and use the resulted values for the aggregation table and the chart diagram [Task 3]

In the result part of the web application, the derived nutrients have to be integrated into the sample enlistment, the aggregation table and the Line-/Scatterdiagram. To compute the data values that are used for the aggregation table and the chart diagram, a PL/pg SQL function will be proposed that is based on the approach described in chapter 5.5.2.

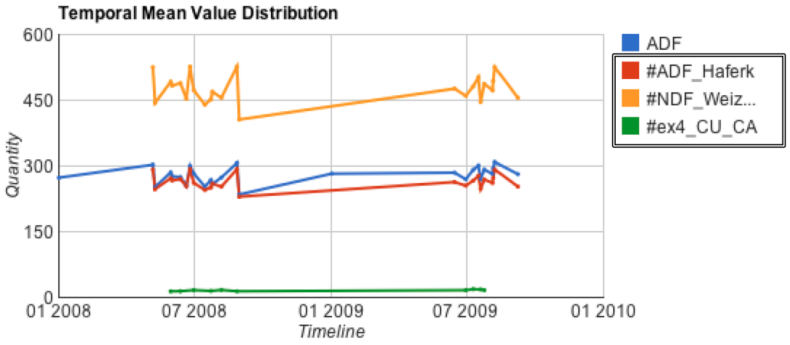


Figure 6.8: Visualization of task 3 concerning the integration of derived nutrients

4. Compute sample results for derived nutrients and use the resulted values for the sample enlistment [Task 4]

In the sample enlistment table on the left side of the web page, only values from the same measurement sample are used for the computation of a derived nutrient. Figure 6.9 shows the sample enlistment with integrated derived nutrient.
Furthermore, the outlier detection functionality should be supported for derived nutrients as well.

	LIMS-Nr.	Canton	PLZ	#ex4_CU_CA	CA	CU
1	XXXXXXX8738483	Aargau	5642		5.825	
2	XXXXXX84-014	Aargau	5722	13.566	5.605	7.961
3	XXXXXX31-007	Bern	2608	13.101	6.663	6.438
4	XXXXXX68-001	Bern	2608	14.58	7.71	6.87
5	XXXXXX62-003	Bern	2612		5.4	

Figure 6.9: Visualization of task 4 concerning the integration of derived nutrients

6.3 Insertion of derived nutrients into the nutrient select field [Task 1]

6.3.1 Creation of table containing formulas

Table 6.1 shows an extract of the relation `t_formulas`. In this table, some derived nutrients are listed with their abbreviation and the corresponding formula. The #-symbol before the abbreviation is used as an indicator for derived nutrients. If in a formula the abbreviation of a component starts with the #-sign, the formula has to be expanded by replacing those abbreviations with the related formula. For instance, the formula of the derived nutrient with id=4 involves the nutrient #OS[g_kg TS] as a component. So, the formula of #OS[g_kg TS] replaces that abbreviation. This replacement procedure is done until the formula contains just non-derived components. In the case of the derived nutrient with id=4 in Table 6.1, it results the following expanded formula: 0.0196 * 1000 - RA[g_kg TS].

Furthermore, it has to be mentioned that some of the derived nutrients are only valid for some specific feed types. In order to compute these nutrients just for the related feed types, a separate table called `t_formula_feed` is defined which contains the specified relations between nutrients and feeds. For example the second row of table `t_formula_feed` implies that the derived nutrient with id=3 is valid for the feed with the `feed_key`=193 which corresponds to the feed named as Soja.

Most of the nutrients, as for instance the one with id=5 are valid for all the feed types. Instead of defining for each feed a separate row in table `t_formula_feed`, the id of the nutrient is listed just once with a corresponding `id_feed` that is 0. A zero-value in the column `id_feed` indicates that the corresponding nutrient is valid for all the feeds.

t_formulas

id	abbreviation_de	formula
1	#ADF_Haferk[g_kg TS]	(0.9559 * RF) + 24.461
2	#NDF_Weizengpfl[g_kg TS]	(-0.0059*(RF*RF))+(4.7608*RF)-345.01
3	#ADF_Sojaschrot[g_kg TS]	(1.3265 * RF) + 21.394
4	#BE_Maisganzpfl[MJ_kg TS]	0.0196 * **#OS[g_kg TS]**
5	#OS[g_kg TS]	1000 - RA[g_kg TS]
10	#ex_ZUCK_ADF	ZUCK + ADF
11	#ex_FE_ZUCK	FE + ZUCK
12	#ex_CU_CA	CU + CA /10
13	#ex_MG_derived	MG + **#ex_CU_CA**

Table 6.1: Table `t_formulas` containing derived nutrients (id 10-13: fictive nutrients)

t_formula_feed

id_formula	id_feed
5	0
3	193
4	1106
4	1104

d_feed

feed_key	name_de
193	Soja
1106	Maisganzpflanze, Teigreife
1104	Maisganzpflanze, Milchreife
800	Haferflocken

Table 6.2: Extract of table `t_formula_feed` and `d_feed`

6.3.2 Query

In a next step, a query has to be defined, that retrieves all the derived nutrient abbreviations from the table `t_formulas` that should be displayed in the nutrient select field. To get all the derived nutrients that should appear in the nutrient field, it has to be iterated on table `t_formulas`. It has to be checked for each nutrient whether there is data for all involved components. The query that is described in this section retrieves all the id numbers of derived nutrients where enough data is available, considering the selected conditions. These conditions are specified in the `sql_from_where`-query that is generated in JavaScript depending on the selected options. The subqueries of the WHERE-clause (line 7-10 and 16-30) are described below:

- The first subquery (line 7 to 10) retrieves all the involved component abbreviations of a derived nutrient. This is done with the help of the function `getInvolvedAbbreviations()`. The array with all the involved abbreviations that is returned by this function can be transformed to a set of rows with the pg-array-function called `unnest(anyarray)`.[5]
- The second subquery selects all the involved abbreviations that are retrieved by the `sql_from_where`-query.

If the total set difference of these two subqueries is empty, data for each component of the formula is available. So, the ID of the corresponding derived nutrient is retrieved, which means that the corresponding nutrient abbreviation will appear in the nutrient field.

Query to retrieve all the IDs of derived nutrients for the nutrient field:

```
1  select id
2  from t_formulas as f1
3  where not exists(
4
5      /** Selects all the involved abbreviations of a formula with a given id. **/
6
7      (
8      select abbreviation
9      from unnest(getInvolvedAbbreviations((select formula from t_formulas as f2 where
           f2.id=f1.id)::text)) as abbreviation
10     )
11
12     except
13
14     /** Selects all the involved abbreviations of a formula with a given id for
           which data is available that satisfies the selected restriction conditions.
           **/
15
16     (
17     select abbreviation
18     from  unnest(getInvolvedAbbreviations((select formula from t_formulas as f2
           where f2.id=f1.id)::text)) as abbreviation
19     where abbreviation in
20
21         (
22         select abbreviation_de
23
24         /** sql_from_where-Query (generated in JavaScript). **/
25
26         from d_nutrient, fact_table, d_time, d_origin, d_quality_parameters, d_feed
27         where id_time_fkey=time_key and id_origin_fkey=origin_key and
               id_quality_fkey=quality_key and id_feed_fkey=feed_key and (
               drying_condition_de in ('unbelüftet')) and (d_feed.name_de in ('Emd'))
28             and nutrient_key=id_nutrient_fkey and t_day is not null
29         )
30     )
31
32 )
```

6.3.3 PHP / JavaScript Implementation

The integration of the previously described query into the web application requires changes in the following PHP- and JavaScript-Files:

- js1.update_selectfield.js
- jsE.update_selectfield_Queries.js
- ajax-pg-options.php

In order to add the retrieved derived nutrient abbreviations to the nutrient select field, the javaScript function `js1_getNewOptions()` was changed as follows: If the select field that has to be updated is named `nutrient[]`, the query that is sent as an ajax request is composed of two parts:

1. The first query part consists of all the **non-derived** nutrients that should appear in the list. This query is stored in the variable `sql_newOptions`.
2. With the query in the variable `sql_newOptions_derived`, the abbreviations of **derived** nutrients are retrieved. This is done with the help of the query that is discussed in the previous subchapter. How this query is generated in JavaScript can be looked up in the function `jsE_updateNutrientField()`.

In order to get all the abbreviations that should appear in the select field, the queryparts named `sql_newOptions` and `sql_newOptions_derived` are put together with the `UNION` operation. The illustration below depicts the basic structure of the query as it is explained above.

Query structure to retrieve all the options for the nutrient select field:

```
/**
   Query, that returns all non-derived nutrient abbreviations that should
   appear in the nutrient select field. The Query is generated in the
   function jsE_updateBaseQuery().
**/

   sql_newOptions
```

UNION

```
/**
   Query, that returns all derived nutrient abbreviations that should appear
   in the nutrient select field. The Query is generated in the function
   jsE_updateNutrientField(sql_from_where)
**/

   sql_newOptions_derived
```

6.4 Update select fields depending on selected derived nutrients [Task 2]

6.4.1 Query

The aim of this task is to retrieve the options for a specific select field, depending on the previously selected conditions. Special consideration is required if a derived nutrient is selected in the selection part of the web application. In such a case, it has to be checked for each component of the derived nutrient, whether there is data available or not.

The query that retrieves the desired options is explained in this section by the example of the select field `canton[]`. For the select fields labeled as `altitude[]`, `year[]` and `season[]`, the query works in the same way.

If a derived nutrient is selected, all the cantons should be displayed where enough data for the computation is available. This means that for each component of the derived nutrient, it has to be checked if there is data available for a specific canton or not. In case there is no data for one or multiple nutrient components of the formula, the canton is not displayed in the options list.

The query that retrieves the desired result can be structured as the one that is described in task 1. This means that it can be stated using the `NOT EXISTS` and the `EXCEPT` operator. Unfortunately, it could be verified, that the query is quite cost-intensive in this case.

An alternative query that retrieves the desired cantons in a more efficient way, is presented in the listing below. The description of the query follows on the next page.[9]

Query to retrieve all cantons that should appear in the canton select field:

```
1  select distinct canton_de /** canton[] is used as an example select field **/
2  from d_origin as o
3  where not exists
4
5      (select *
6      from d_nutrient as n
7      where (d.abbreviation_de in
8              (select abbreviation from unnest(getInvolvedAbbreviations((select
                   formula from t_formulas where abbr='#ADF_Haferk')::text)) as
                   abbreviation
9              )
10         )
11         and not exists
12
13         (select *
14
15         /** sql_from_where-Query (retrieves selected options) **/
16         from d_nutrient, fact_table, d_time,d_origin, d_quality_parameters, d_feed
17         where id_time_fkey = time_key and id_origin_fkey = origin_key and
               id_quality_fkey = quality_key and id_feed_fkey = feed_key and (
               drying_condition_de in ('belüftet')) and (d_feed.name_de in ('Emd')) and
                nutrient_key=id_nutrient_fkey
18
19         /** additional restriction: There must be a timestamp for the measurement
               value, otherwise it cannot be used for the time-varying computation  of
               the regression. **/
20         and t_day is not null
21         and  o.canton_de=canton_de and n.abbreviation_de=abbreviation_de)
22     )
```

This alternative query uses two-level nesting and is more complex than the one that is defined in task 1. In the part from line 13 to 17, all the tuples that statisfy the selected restriction conditions are retrieved. This query is generated in JavaScript and stored in a variable called `sql_from_where`. The listing above considers the case, that the feed "Emd" and the drying condition "belüftet" is chosen in the selection part of the web application.

The outer nested query (from line 5) selects any `d_nutrient` tuples with a nutrient abbreviation that corresponds to one of the abbreviations that is involved in the formula, if there is not a tuple with the same abbreviation in the retrieved result of the generated `sql_from_where`-query.

So, the whole query can be rephrased as follows: Select each canton such that there does not exist an abbreviation of the derived nutrient components that is not in one of the tuples that are retrieved by the `sql_from_where` query.

To put more simply: A canton is retrieved if there is data available (with the selected restriction condition) for each involved nutrient component of a formula.

6.4.2 PHP / JavaScript Implementation

The integration of the previously described query into the web application requires changes in the following PHP- and JavaScript-Files:

- js1.update_selectfield.js
- jsE.update_selectfield_Queries.js
- ajax-pg-options.php

The query to get all the options for the select field (that is followed by the nutrient field) is generated in the the function `jsE.updateAfterNutrientField()` and consits of two parts. If we take the select field canton as an example field that has to be updated, the structure of the query would look like that:

1. In the first part of the query, all the cantons are retrieved where non-derived measurement values exist.

2. In the second part of the query, all the cantons are retrieved, where we have enough data to compute the selected derived nutrient. This second query part is also generated in the function `jsE.updateAfterNutrientField()`.

As in task 1, both query parts are put together with the UNION command the retrieve all the options for a specific nutrient field. The structure of the query is depicted below:

Query structure to select all the options for a specific select field:

```
/**
    Query that retrieve all
    [canton || altitude || year || season],
    where exists data for the selected "non-derived" nutrients.
    (generated in function jsE_updateAfterNutrientField())
**/
```

UNION

```
/**
    Query that retrieve all
    [canton || altitude || year || season],
    where exists data for the selected derived nutrients.
    (generated in function jsE_updateAfterNutrientField())
**/
```

6.5 Compute time-varying regressions [Task 3]

6.5.1 PL/pg SQL Function for temporal results

The PL/pg SQL function that presented in this chapter is used for the integration of derived nutrients into the result display of the web application. It is based on the PL/pg SQL function that makes use of a 3-dimensional array to store the relevant measurement data in memory. This approach was presented in chapter 5.5.2.

The function is declared as`computederivedresults(expression, sql_from_where)`:

- The first input parameter corresponds to the formula of the selected derived nutrient.
- The second parameter that is named as `sql_from_where` is the query that is generated in JavaScript depending on the selected options.

The basic structure of the PL/pg SQL Function corresponds to the already presented one in chapter 5.5.2. However, the main steps of the function is explained below.

Step 1: Expand the formula if it contains derived components (indicated by a #-sign).

Step 2: Extract all abbreviations from the formula and store them in an array.

Step 3: Fill a 3-dimensional array with data from the database.

Step 4: Perform combining algorithm as described in chapter 5.5.2.

Step 5: Replace all the abbreviations that are in the formula with the corresponding measurement values that are resulted from the combining process.

Step 6: Compute the result of the formula (formula contains now only numerical values because of the replacement in step 5).

Step 7: Return the computed value and the timestamp that was considered as fix.

6.5.2 PHP / JavaScript Implementation of temporal results

The integration of time-varying derived nutrients values into the web application requires changes in the following PHP- and JavaScript-Files:

- js2_result_coordinator.js
- js4_result_diagram.js
- js4_result_aggregate.js
- jsF_resultQueries.js
- ajax-pg-result-aggregate.php
- ajax-pg-result-linediagram.php
- ajax-pg-result-scatterdiagram.php

The query that retrieves the computed values of a selected derived nutrient is generated in the javaScript-function `jsF_createResultQuery_derived(sql_from_where)`. With the UNION operation that query is set together with the query that retrieves all the result values for the selected non-derived nutrients. If there are for instance only non-derived nutrients selected, only the first part of the query is sent as an ajax-request to the server. The same applies the other way round.

The illustration below shows the basic structure of the query that is sent to the server to get the required data for the aggregation table and the line-diagram.

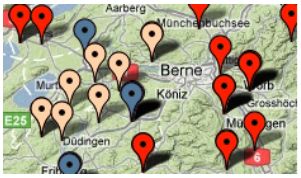
Query structure to select values used in the aggregation table:

```
(select abbreviation_de, count(quantity), min(quantity), max(quantity), avg(
    quantity), stddev_samp(quantity)

   /* input: sql_from_where variable from jsF_createResultQuery_nonderived */

group by abbreviation_de order by abbreviation_de)
```

UNION

```
(select allderived.abbreviation_de, count(allderived.quantity), min(allderived.
    quantity), max(allderived.quantity), avg(allderived.quantity), stddev_samp(
    allderived.quantity)
from(

   /* input: sql_from_where variable from jsF_createResultQuery_derived */

   ) as allderived
group by allderived.abbreviation_de order by abbreviation_de))
```

Visualization of aggregation table containing a derived nutrient:



Nutrient	Minimum	Maximum	Average	2σ*	Count	avg(μ< -σ)	avg(μ±σ)	avg(μ> +σ)
ZUCK	58.000	194.000	113.037	21.514	215	80.467	111.497	146.556
#ADF_Haferk	228.068	293.069	260.777	15.752	32	231.967	259.377	290.969

* Standard deviation of a population

Note: A #-symbol before the abbreviation indicates derived nutrient parameters that are computed temporal. Data with unspecified measurement time are obmitted for the computation.

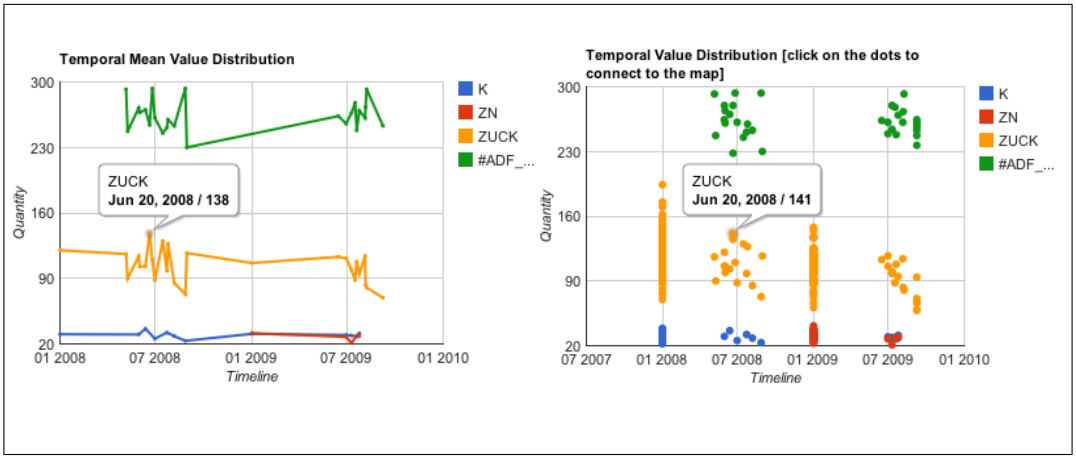
Query structure to select values used in the line-diagram:

```
((select abbreviation_de, (case when t_day is null then to_date(t_year||'-01-01'
    , 'YYYY-MM-DD') else t_day end) as day, t_year, avg(quantity)

   /* input here: sql_from_where variable that is generated in
       jsF_createResultQuery_nonderived */

) group by abbreviation_de, t_day, t_year order by abbreviation_de, day)
```

UNION

```
((select allderived.abbreviation_de, to_date((allderived.day::text),'YYYYMMDD')
    as t_day, null, avg(allderived.quantity)
from(

   /* input here: sql_from_where variable that is generated in
       jsF_createResultQuery_derived */

   ) as allderived
group by allderived.abbreviation_de, t_day order by abbreviation_de, t_day)
```

Visualization of line-diagram/scatter-diagram containing a derived nutrient:



6.6 Compute derived nutrient values grouped by measurement samples [Task 4]

6.6.1 Query

For the sample enlistment table, only values from the same meaurement sample are used for the computation. The listing below shows how the query can be defined to retrieve the desired measurement values that are later used for the calculation.

Assume that we have a derived nutrient #ZUCK_FE containing the components ZUCK and FE. In this case and we want to list all the sample keys with the corresponding ZUCK and FE-Values. So, the formula can be calculated with the values that come from the same measurement sample. For that, the `sample_key` and an array with all the nutrient component values that correspond to the `sample_key` is retrieved. Is there no ZUCK value or no FE value in a specific measurement sample, the `sample_key` is not retrieved. In case that there are multiple measurement values of a nutrient in the same sample, the average of all these values is taken.

Query that retrieves all the sample keys with their related ZUCK and FE value

```
1  select array[ZUCK.quantity, FE.quantity] as values, sample_key
2  from
3      (select avg(quantity) as quantity, id_sample_fkey
4      from d_nutrient,
5
6          /* insert here sql_from_where-Query according to selected options */
7
8          fact_table, d_time, d_origin, d_quality_parameters, d_feed
9      where id_time_fkey = time_key and id_origin_fkey = origin_key and
           id_quality_fkey = quality_key and id_feed_fkey = feed_key and (
           drying_condition_de in ('belüftet')) and (d_feed.name_de in ('Heu')) and
           nutrient_key=id_nutrient_fkey and d_nutrient.abbreviation_de ='ZUCK'
10     group by id_sample_fkey) as ZUCK,
11
12
13     (select avg(quantity) as quantity, id_sample_fkey
14     from d_nutrient,
15
16         /* insert here sql_from_where-Query according to selected options */
17
18         fact_table, d_time, d_origin, d_quality_parameters, d_feed
19     where id_time_fkey = time_key and id_origin_fkey = origin_key and
           id_quality_fkey = quality_key and id_feed_fkey = feed_key and (
           drying_condition_de in ('belüftet')) and (d_feed.name_de in ('Heu'))
20
21     and nutrient_key=id_nutrient_fkey and d_nutrient.abbreviation_de ='FE'
22     group by id_sample_fkey) as FE,
23
24     d_sample
25
26 where ZUCK.id_sample_fkey=sample_key and FE.id_sample_fkey=sample_key;
```

- In the SELECT-clause of the query, the `sample_key` is retrieved with a corresponding array that contain values for all the nutrient components for a specified derived nutrient. In the example above, the derived nutrient is composed of the nutrients ZUCK and FE.
- In addition to the relation `d_sample` that occurs in the FROM-clause, a temporary table is defined for each involved component (line 3-10 and 13-22). In these subqueries, all the measurement values (`quantity`) that satisfy the restriction condition that are selected in the select part of the web application. Together with those measurment values, the corresponding sample keys are retrieved.

- In the WHERE-clause, the join condition on the `sample_key` is set. So, all the values that are retrieved as an array are measured in the same sample with the corresponding `sample_key`. With the help of an array as an attribute in the select clause, it is guaranteed that the query works for a various number of nutrient components.

In the next sub-chapter it is explained, how the described query is generated dynamically depending on the selections in the web application.

6.6.2 PL/pg SQL Function for sample results

The PL/pg sql function `computederivedresults_bySamples(formula, sql_from_where)` calculates derived nutrients values by using only measurement values that come from the same measurement sample. As input parameter of this function acts the formula of a specific derived nutrient and the `sql_from_where`-Query that represents the selected options in the web application. The main steps in these functions are listed as follows:

1. In a first step, all the abbreviations that are involved in a derived nutrient must be retrieved. This is done with the help of the function `getInvolvedAbbreviations(formula)`, which returns an array with all the abbreviations that are in the specified formula.

2. In a second step, a query as described in chapter 6.6.1 is generated. The retrieved result of that query contains all the measurement values of the involved nutrient components, grouped by the `sample_key`. The listing below shows the generation of that query, where for each abbreviations in the array `abbr[]` , an additional part in the select-clause, from-clause, and where-clause is added.

```
FOR n in array_lower(abbr, 1) .. array_upper(abbr, 1) LOOP

    sql_select  := sql_select || ', '|| abbr[n] ||'.quantity';

    sql_from    := sql_from ||
            '(select avg(quantity) as quantity, id_sample_fkey
            from d_nutrient, '

            || sql_from_where ||

            ' and nutrient_key=id_nutrient_fkey and d_nutrient.abbreviation_de
                = ' || ''''' || abbr[n] || ''''''|| '
            group by id_sample_fkey) as '|| abbr[n] ||',';
    sql_where   := sql_where || ' and ' ||  abbr[n]|| '.id_sample_fkey=
        sample_key';

end loop;

sql_query :=    'select array[' || substring(sql_select,2) || '] as a,
    sample_key
        from ' || sql_from || ' d_sample' ||
        ' where ' || substring(sql_where, 5);
```

3. Then, all the abbreviations of the formula are replaced with the measurement values from the query and the result of the expression is computed.

4. Finally, the function returns for each sample the result of the calculation and the corresponding `lims_number` that refers to a specific `sample_key`.

6.6.3 PHP / JavaScript Implementation for sample results

The integration of derived nutrients values into the sample enlistment of the web page requires changes in the following PHP- and JavaScript-Files:

- js2_result_coordinator.js
- js3.result_list-map.js
- jsF.resultQueries.js
- ajax-pg-result-google.php

The illustration below show how to basic structure of the query to retrieve the required data for the sample enlistment. Again, the query is split in two parts:

1. In the first part, the measurement values for **non-derived** nutrients are retrieved, together with the corresponding `lims_number` and the nutrient abbreviation.
2. In the second part, the calculated values of **derived** nutrients are retrieved. The `lims_number` that indicates the measurement sample is retrieved as well. The second part of the query is generated in `jsF_createResultQuery_derived_bySamples()`.

Query structure to select values used in the sample enlistment:

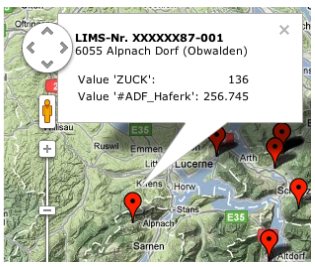
```
(select distinct lims_number, abbreviation_de, quantity

from
   /* sql_from_where, created in jsD_CreateFromWhereQuery() */

order by lims_number)
```

UNION

```
(select allderived.lims_number, allderived.abbreviation_de, allderived.quantity
    as quantity

from(
   /* sql_derived_bySamples, created in jsF_createResultQuery_derived_bySamples(
        sql_from_where) */

) as allderived

order by abbreviation_de, lims_number)
```

Visualization of sample enlistment with a derived nutrient:

	LIMS-Nr.	Canton	PLZ	#ADF_Haferk	K	ZN	ZUCK
1	XXXXXX81-001	Appenzell Innerrhoden	9050	229.024			141
2	XXXXXX49-001	Appenzell Innerrhoden	9050	236.671	31.1		144
3	XXXXXX29-001	Appenzell Innerrhoden	9050	249.098	29.629		142
4	XXXXXX64-001	Appenzell Innerrhoden	9054	234.759			147
5	XXXXXX87-001	Obwalden	6055	256.745			136
6	XXXXXX40-001	Schwyz	6403	248.142			
7	XXXXXX28-006	Schwyz	6403	250.053			137
8	XXXXXXXXX343475375	Schwyz	6417	238.164			
9	XXXXXX53-001	Schwyz	6417	238.583			
10	XXXXXX12-001	Schwyz	6417	269.171			96
11	XXXXXX55-001	Schwyz	6417	255.789			117



Chapter 7

Summary

In the preceding elaborations, some implementation methods for computing regressions for derived nutrients have been proposed. The focus of the project has been to find an efficient solution for computing derived nutrients as time-varying regressions. For that, different approaches have been described and implemented. Because of the performance problems described in chapter 5.2.3, the approach of using standard SQL to retrieve the computed values of derived nutrients could not be carried out successfully. In any case, an approach using SQL views is not scalable. This means that it is not possible to compute time-varying values on a variable data set that is selected by the user, because the data set has to be specified in the view definition. That is why an alternative solution had to be found.

Finally, a number of algorithms were implemented as PL/pg SQL functions. The most efficient approach, which involves using 3-dimensioal arrays (presented in chapter 5.5.2) was then used for the integration of derived nutrients into the current web application.

The adaption of the web application consisted of four main tasks, which are dividable into two main stages: After the changes of the update mechanism accoridng to the selection fields has been made, the methods that retrieve and display the result data have been adapted.

For derived nutrients, two different computation methods were integrated:

- First, the derived nutrient values are computed as time-varying regressions. The computed **temporal result values** are used for the representation of the line-diagram and the scatter-diagram. The aggregation values for displaying statistical information about a derived nutrient are computed by the same algorithm. With the help of the line-diagram, the derived nutrient values can be compared over time and historical analyses are possible.

- On the other hand, derived nutrients can be computed as **sample result values.** These values are only calculated for measurement samples that contain data for each component involved in a formula. These values are integrated in the sample enlistment on the left hand-side of the wep-page.

Both methods support the calculation of any type of formula that is composed of algebraic expressions containing the most common mathematical operators. The PL/pg SQL function has been tested for some sample formulas of derived nutrients on the complete data set. So far, the web-application has been tested on a limited data set. Depending on the complexity of the formula and the number of tuples has to be retrieved and stored in memory, the execution time can amount to several seconds.

An possible approach to the solution would be the development of a restriction condition, limiting the number of selectable derived nutrients. Otherwise, a potential execution time of several seconds has to be put up.

An alternative and more efficient solution could be realized by computing the regression results for the historical representation in the same way as it is done in the sample enlistment. In that case, the derived nutrient values are computed only with data that exists within a measurement sample. However, this would decrease the number of data values, limiting the significance of the results considerably. On the other hand, the use of the cost-intensive combining algorithm would not be needed, which would result in an improvement of performance.

Bibliography

[1] Agroscope Liebefeld-Posieux (2009). Schweizerische Futtermitteldatenbank. URL: *http://www.agroscope.admin.ch/futtermitteldatenbank/index.html?lang=de* [Last access: 2011-11-13]

[2] Douglas, K., Dougklas S. (2003) PostgreSQL. A comprehensive guide to building, programming, and administering PostgreSQL databases. Sams Publishing. First Edition.

[3] Kemper, A. , Eickler, A. (2006). Datenbanksysteme: Eine Einführung. Oldenbourg Wissenschaftsverlag. 6. Auflage.

[4] Gupta A. , Mumick S. (1999). Materialized views techniques, implementations, and applications. MIT Press. First Edition.

[5] The PostgreSQL Global Development Group (2010). PostgreSQL 9.0.4 Documentation. URL: *http://www.postgresql.org/docs/9.0/interactive/index.html* [Last access: 2011-11-20]

[6] Hitoshi, H. (2008). Window Functions for PostgreSQL Design Overview. URL: *http://umitanuki.net/pgsql/wfv08/design.html*

[7] Worsley J., Drake J. (2002). Practical PostgreSQL. URL: *http://www.linuxtopia.org/online_books/database_guides/Practical_PostgreSQL_database* [Last access: 2011-11-20]

[8] Google (2011). Google chart tools. Collection of tutorial, examples and API library URL: *http://code.google.com/intl/en/apis/chart/interactive/docs/index.html* [Last access: 2011-11-22]

[9] Elmasri R., Navathe S. (2004). Fundamentals of Database Systems. Pearson Education. Fourth Edition.