# Implementation of an Aggregating Nearest Neighbors Join Technique in the Swiss Feed Database

**Master Project in Computer Science**

**Samuele Zoppi 08-702-664, Francesco Luminati 07-710-387**

Zurich, Switzerland

**Institute for Informatics**

**University of Zurich**

**Prof. Dr. M. Böhlen**

Supervisor: Francesco Cafagna

# Contents


**1 Introduction** **5**

**2 Problem Description** **7**
2.1 Derived Nutrients . . . . . 7
2.2 Storage of Formulas and Measurements . . . . . 7
2.3 Computation of Formulas . . . . . 8

**3 Solutions** **11**
3.1 SQL Approaches . . . . . 11
3.1.1 Brute Force SQL . . . . . 11
3.1.2 Optimized SQL ($SQL^+$) . . . . . 12
3.2 A-NNJ Algorithm . . . . . 13

**4 Experimental Evaluation** **19**
4.1 Setup . . . . . 19
4.2 Ties Management . . . . . 19
4.3 Selectivity of Condition $K = K^*$ . . . . . 21

**5 Visualization of the Results** **25**
5.1 Online Interface . . . . . 25
5.2 PHP implementation . . . . . 26

**6 Conclusions** **31**

**Bibliography** **33**

# 1 Introduction

The computation of derived nutrients values in the Swiss Feed Database is an important and challenging task. The automatized computation of derived nutrients has to ensure the same quality as a manual computation. Since nutrients measurements are not taken regularly, a nearest neighbor(s) search technique has to be used to compute the derived values for a given timestamp; successively, in case more than one nearest neighbor exists, all the nearest neighbors have to be aggregated in order to ensure the best quality in the result. Derived nutrients values are calculated newly for each query and cannot be stored in the database since new measurements are entered every day and since the user queries are subject to temporal and geographical restriction changes. The computation of derived nutrients requires that the formulas, that are used to infer the derived nutrients values, are stored in the database.

In order to accomplish our task, two approaches have been explored: on the one hand, a plain SQL solution ($SQL^+$) has been implemented following the idea of the KNN-join technique described in [2], and on the other hand, an aggregating nearest neighbors join technique (A-NNJ) has been implemented as a function using the procedural language PL/pgSQL.

We experimentally evaluated our approaches with two different tests. The first experiment tests the ability of dealing with ties in the nearest neighbors, i.e. in the cases that for a given timestamp two or more nearest neighbors are found. The second experiment aims to test the performances of both approaches changing the number and the distribution of different nutrients in the dataset. In both experiments the A-NNJ algorithm yields better performances in almost all the cases; the SQL solution only performs better when there are no ties at all or when the dataset is merely composed of measurements of the searched nutrient.

Derived nutrients values can be visualized online, since the A-NNJ technique has been integrated into the online interface of the Swiss Feed Database.

This document is organized as following. Section 2 contains the problem description and the challenges in the computation of derived nutrients. Section 3 describes the possible solutions to our task. The experimental evaluation is reported in Section 4. Section 5 presents the online interface and the details about the integration of the results into it. Section 6 states the conclusions for this project.

# 2 Problem Description

## 2.1 Derived Nutrients

The Swiss Feed Database stores values for more than 400 nutrients on about 400 different feeds. We can distinguish between measured values and computed ones. Measured values are taken from laboratory analysis on feeds samples and are stored in the database, while computed values are calculated using formulas. A formula infers the value of a certain nutrient using the value of other nutrients. For instance, we can infer how much organic matter is in a certain feed, from the quantity of his ashes after having burned it. From now on, nutrients, whose values are calculated using a formula, will be called derived nutrients.

**Example:** If we take a sample of 1000 grams of dried corn, we burn it, and the remaining ashes weigh 14 grams, then the quantity of organic matter can be inferred using the formula: "dried sample weight" - "ashes weight" = 1000 g - 14 g = 986 g.

Formulas can depend on one to many other measured as well as derived nutrients. The goal of this project is to find a way to automatize the computation of derived nutrients. More about the complexity of this process is described in the following sections.

## 2.2 Storage of Formulas and Measurements

The formulas have been stored in a table as strings and with their abbreviation as shown in Table 2.1. The abbreviation of the formula is actually the name of the derived nutrient, which is inferred using the formula. In case a derived nutrient A is calculated using another derived nutrient B, the abbreviation of B will be used inside the formula of nutrient A, e.g. row number 2 of Table 2.1.

| id | abbreviation | formula |
|---|---|---|
| 1 | #OS[g_kg TS] | 1000 - RA[g_kg TS] |
| 2 | #OS[g_kg FS] | TS[g_kg] * #OS[g_kg TS] / 1000 |
| 3 | #NDF_Weizenk[g_kg TS] | (1.3374 * RF[g_kg TS]) + 94.979 |

Table 2.1: Formulas table

The abbreviation of each formula begins with a hash mark (#), so that we can recognize if a formula contains other formulas or just measured nutrients. A PL/pgSQL function has been developed in order to expand a formula until it does not contain any other formula.

**Example:** The not expanded version of the formula #OS[g_kg FS] is TS[g_kg] * #OS[g_kg TS] / 1000, its expanded version is TS[g_kg] * (1000 - RA[g_kg TS])/ 1000, see Table 2.1.

Nutrient's measurements are stored and historized. So, for each combination of feed and nutrient, several measurements are stored sparse in time. This means, that measurements are not taken regularly, so we do not have any guarantee about finding a measurement for a certain nutrient of a certain feed on a given timestamp; if we want to compute the value of a formula on a given timestamp, we have to find the temporal nearest value(s) (in respect to the given timestamp) of each nutrient involved in the formula.

As a second challenge the value of a derived nutrient has to be computed each time newly and cannot be stored: since new values for measured nutrients can be entered anytime into the database and since the dataset on which the formula are calculated can change respect to temporal and geographical restriction, so the nearest neighbor(s) can change from time to time. A formal description of the computation of formulas is given in the next section.

## 2.3 Computation of Formulas

Given a table $R$ storing tuples of feeds $F$, swiss cantons $C$ and timestamps $T$ and a table $S$ storing tuples of feeds $F$, swiss cantons $C$, timestamps $T$, nutrients $K$ and measurements $M$. We want to compute the value of a formula for each row of $R$. This means, for each row of $R$, find the temporal nearest neighbor(s) in $S$ for each nutrient $K$ in the formula, where the feed type and the canton correspond to the ones in $R$. The feed name $F$ and the swiss canton $C$ are taken as a single group condition named $G$. If $S$ has more than one nearest neighbor, then we take the average among all nearest neighbors.

| Notation | Meaning | Example |
|---|---|---|
| $R$ | A table storing tuples of timestamps and feeds | {(01.01.2010,"Corn"), (20.12.2011, "Barley")} |
| $S$ | A table storing tuples of timestamps, feeds, nutrients and measurements | {(01.01.2010, "Corn", "RA", 3.34), (20.12.2011, "Barley", "OS", 2.55)} |
| $r$ | A tuple of $R$ | (01.01.2010,"Corn") |
| $s$ | A tuple of $S$ | (01.01.2010, "Corn", "RA", 3.34) |
| $M$ | A measured value | 3.34 |
| $K$ | A nutrient name | Potassium |
| $K'$ | A derived nutrient name | #OS |
| $F$ | A feed name | Corn |
| $T$ | A timestamp | 01.01.2010 |
| $C$ | A swiss canton name | Zurich |
| $G$ | A group condition | ($F$, $C$) |
| $Z$ | A result table storing tuples of feeds, cantons, timestamps, derived nutrients and computed measurements | {("Corn", "Zurich",01.01.2010, "#OS",4.45), ("Barley", "Zurich",20.12.2011, "#OS",4)} |

Table 2.2: Notation

**Example:** Given the formula #A = B + C and $R$ and $S$ as in Table 2.3. The values for the derived nutrient #A for each row in $R$ are reported in Table 2.4.

(a) Table $R$

| $F$ | $C$ | $T$ |
|---|---|---|
| Corn | Zurich | 22.10.2008 |
| Barley | Aarau | 14.5.2006 |
| Corn | Bern | 04.08.2011 |

(b) Table $S$

| $F$ | $C$ | $T$ | $K$ | $M$ |
|---|---|---|---|---|
| Corn | Zurich | 03.10.2008 | B | 2.5 |
| Corn | Bern | 12.12.2011 | B | 3.5 |
| Barley | Aarau | 17.08.2006 | B | 5.7 |
| Barley | Aarau | 17.08.2006 | B | 3.3 |
| Corn | Zurich | 21.10.2008 | C | 7.4 |
| Corn | Zurich | 23.10.2008 | C | 9.6 |
| Barley | Aarau | 17.02.2007 | C | 2.5 |
| Corn | Bern | 04.08.2011 | C | 3.5 |

Table 2.3: Example of $R$ and $S$

| $F$ | $C$ | $T$ | Value of #A = B + C |
|---|---|---|---|
| Corn | Zurich | 22.10.2008 | 2.5 + (7.4 + 9.6) / 2 = 11 |
| Barley | Aarau | 14.5.2006 | (5.7 + 3.3) / 2 + 2.5 = 7 |
| Corn | Bern | 04.08.2011 | 3.5 + 3.5 = 7 |

Table 2.4: Table $Z$: values for the derived nutrient #A

The value for #A for corn in canton Zurich on 22.10.2008 (first row in Table 2.4) is calculated taking the temporal nearest value of nutrient B in $S$ for corn in Zurich, that is the value of B on 03.10.2008. For C we have two nearest neighbors, the value on 21.10.2008 and the value on 23.10.2008, for this reason we take the average between them.

In the next section we are going to present an algorithm, which automatizes this process. As an additional constraint, we do not allow values in $S$ to be nearest neighbors, if their distance from the timestamp in $R$ is more than 365 days, since the values of measured nutrients can change substantially during a year.

# 3 Solutions

In this chapter we are going to present three different approaches to automatize the computation of formulas. First, we investigate two plain SQL queries, a Brute Force method and an optimized SQL solution ($SQL^+$). After that, we present one solution using the procedural language PL/pgSQL (the A-NNJ algorithm). Throughout this chapter, the notation of Table 2.2 is used.

## 3.1 SQL Approaches

### 3.1.1 Brute Force SQL

The Brute Force approach derives from the solution proposed by Böhm et al. in [1] and is shown here. For each tuple in $R$ the temporal distance to every tuple in $S$ of nutrient $K^*$ is calculated. The tuple(s) with the smallest distance are the nearest neighbor(s) of the given $K^*$. The query selects group $G$, timestamp $T$ and the temporal nearest nutrient $K'$ in $S$ of each tuple in $R$. The nearest neighbors are found with the sub-query from line 3 to 8 seen in the listing below. The sub-query orders by distance every tuple in $S$ with the same group as $r$ and takes the first, using the *LIMIT*, which is the one with the smallest distance.

The *GROUP BY* operator (line 7) has been introduced in order to deal with the cases where we have more than one nearest neighbor (see Section 2.3): it groups all tuples in $S$ with the same distance to $r$ and applies the average aggregate function $avg()$ on them (line 3).

```
1  SELECT G, T, (
2
3          SELECT avg(M)
4          FROM S
5          WHERE S.G = R.G
6          AND S.K = K*
7          GROUP BY ABS(S.T - R.T)
8          ORDER BY ABS(S.T - R.T) LIMIT 1 ) AS K'
9
10 FROM R
```

The computation of the temporal distance (line 7) is the essential part of this query and must be calculated for all pairs of tuples, which means that for the tuples in $R$ all the tuples in $S$ have to be read from the disk. This operation is very expensive in terms of performance and leads to a quadratic complexity of the query, which is not suitable for large datasets. At the time,

the Swiss Feed Database contains about 3.5 million of tuples; for this reason we discarded this approach from the beginning.

### 3.1.2 Optimized SQL ($SQL^+$)

An optimized SQL solution is introduced by Yao et al. in [2] and is extended in this project to deal with ties. This approach is called $SQL^+$. The idea behind this approach is to reduce as much as possible the computation of distances that we know to be very expensive, since for this calculation we need to read the tuples from the disk. In order to reduce the distance computations we first identify the best nearest neighbor candidates and we then calculate the distance only from them. For each row $r$ of $R$ the best nearest neighbor candidates are those with the biggest timestamp preceding $r$ and the smallest timestamp following $r$. Therefore, the computation of distances is done only on those candidates.

```
1  SELECT G, T, (
2
3  (SELECT avg(S.M)
4          FROM S WHERE R.G = S.G AND S.K = K*
5          AND S.T IN (
6                  (SELECT max(T.T) FROM S AS T
7                   WHERE R.G = T.G AND T.K = K*
8                   AND  T.T < R.T),
9                  (SELECT min(T.T) FROM S AS T
10                  WHERE R.G = T.G AND T.K = K*
11                  AND  T.T >= R.T))
12         GROUP BY ABS(S.T - R.T)
13         ORDER BY ABS(S.T - R.T) LIMIT 1)
14 ) AS K'
15
16 FROM R
```

The listing above shows the solution in a concrete SQL query. Similar to the Brute Force, the query selects group $G$, timestamp $T$ and the temporal nearest nutrient $K'$ in $S$ for every tuple in $R$. The nearest neighbors of the searched $K^*$ are found with the sub-query from line 3 to 14. This time, however, the sub-query is not going to order by distance every tuple in $S$, but only two candidates sets, selected using the *WHERE timestamp IN* construct of line 5, which contains two further sub-queries. From line 6 to 8, the biggest timestamp of all timestamps in $S$ preceding $r$ is selected; from lines 9 to 11 the smallest timestamp of all timestamps in $S$ following $r$ is selected. After this, the sub-query calculates the distance between the two subsets of $S$. The query uses the *GROUP BY* operator (line 12) to deal with ties, in the case that both points have the same distance to $r$ or that the subsets are composed by many tuples with the same timestamp, applying the average function on

them (line 3). Finally the nearest point is selected by the *ORDER BY* and the *LIMIT* operators (line 13).

The strength of this query is given by the fast search of the nearest neighbor candidates when in $S$ an index is put on $G$ and $T$.

## 3.2 A-NNJ Algorithm

Another approach to the computation of derived nutrients has been implemented in the form of a function using the procedural language PL/pgSQL. The function, called A-NNJ($R$,$K'$), is the implementation of a sort merge solution and returns a table $Z$ in which a value of the derived nutrient $K$' for every timestamp in $R$ is computed. The computation consists in finding the nearest neighbor(s) of all nutrients $K$ involved in the formula associated to $K$', aggregate them if necessary, and evaluate the formula to return the value. An example of this procedure was exposed in Section 2.3.

The idea behind this algorithm is to sort the relations in a way that permits to perform only one sequential scan on them to find all the join matches. The sorting attributes are the crucial part and consist of the attributes used by the equijoin and the attributes used by the nearest neighbor join. This idea is explained using the example of Section 2.3. The attributes for the equijoin are in this case the feed $F$ and the canton $C$; the attribute for the nearest neighbor join is the timestamp $T$.

(a) Sorted Table R

| *F* | *C* | *T* |
|---|---|---|
| Barley | Aarau | 14.05.2006 |
| Corn | Bern | 04.08.2011 |
| Corn | Zurich | 22.10.2008 |

(b) Sorted Table $S_1$ for $K = B$

| *F* | *C* | *T* | *M* |
|---|---|---|---|
| Barley | Aarau | 17.08.2006 | 5.7 |
| Barley | Aarau | 17.08.2006 | 3.3 |
| Corn | Bern | 12.12.2011 | 3.5 |
| Corn | Zurich | 03.10.2008 | 2.5 |

(c) Sorted Table $S_2$ for $K = C$

| *F* | *C* | *T* | *M* |
|---|---|---|---|
| Barley | Aarau | 17.02.2007 | 2.5 |
| Corn | Bern | 04.08.2011 | 3.5 |
| Corn | Zurich | 21.10.2008 | 7.4 |
| Corn | Zurich | 23.10.2008 | 9.6 |

Table 3.1: Tables $R$ and $S$ used by the algorithm A-NNJ()

**Example:** As a running example, table $R$ and table $S$ of Table 2.3 are sorted by $F$, $C$ and $T$ resulting in the tables of Table 3.1. Table 3.1b refers to tuples of nutrient $B$, Table 3.1c refers to tuples of nutrient $C$. Imagine now

a pointer on the first line of all three tables. In $R$ (Table 3.1a), the pointer points to the group (*Barley*, *Aarau*) with timestamp 14.05.2006, in $S_1$ (Table 3.1b) it points to the group (*Barley*, *Aarau*) with timestamp 17.08.2006, in $S_2$ (Table 3.1c) it points to the group (*Barley*, *Aarau*) with timestamp 17.02.2007.

At this point, for each $S$ we need to check that the next row in the table has a bigger distance to the pointer in $R$ or an alphabetically bigger group; if this is not the case, the pointer must move down one position and check again. In $S_2$ the next row has an alphabetically bigger group, so the pointer does not move. In $S_1$ instead, the next row has the same group and the same distance, so the pointer moves of one position down; the next row has now an alphabetically bigger group. At this moment, the pointers of $S_1$ and $S_2$ point to the nearest neighbors of the first row of $R$ and the formula can be evaluated and put in the results table $Z$ (Table 3.2). In $S_1$ we have the case of a tie, i.e. the same group with the same distance. In this case, the previous elements must be stored in order to calculate the average value for the computation of the formula. At this point, the pointer in $R$ moves one position down (second row in $R$) and finds the group (*Corn*, *Bern*) with timestamp 04.08.2011. Also the pointers of $S_1$ and $S_2$ move one position down and check the next element; both do not move and the value for the formula is computed. The pointer in $R$ goes to the last row and the pointers of $S_1$ and $S_2$ move another position down. After that the condition check is done; the pointer on $S_1$ does not move, the one on $S_2$ instead has the same group and the same distance with the next element, so it moves one position. The formula is computed and for $S_2$ the average value of the last two rows is used.

| *F* | *C* | *T* | *K'* | *M* |
|---|---|---|---|---|
| Barley | Aarau | 14.05.2006 | #A | 7 |
| Corn | Bern | 04.08.2011 | #A | 7 |
| Corn | Zurich | 22.10.2008 | #A | 11 |

Table 3.2: Results table $Z$ for the formula $\#A = B + C$.

The running example shows that for each tuple in $R$ we can find a join match in $S$ only moving the cursors down in the tables. All this is possible only after having sorted all the tables, which have a total complexity of $O(n * log(n))$.

The function A-NNJ($R$, $K$'), listed in Algorithm 1, requires two inputs parameters: a relation $R$ in form of an SQL statement, and a derived nutrient $K$', e.g. *#OS[g_kg FS]*. The results table $Z$ contains an entry of $F$, $C$, $T$, $K$', $M$ (computed value for the derived fact) for each distinct $F$, $C$, $T$ in $R$.

**Algorithm 1**: a-nnj(R, K')

**Input**: an SQL statement $R$
a derived nutrient $K'$
**Result**: $z$

```
1  begin
2      X ← getFormula(K')
3      R ← SELECT DISTINCT F, C, T FROM R
4            WHERE K IN (K_1, ..., K_l) ORDER BY F, C, T
5      foreach K_j ∈ X do
6          S_j ← SELECT F, C, T, M FROM R
7                WHERE K=K_j ORDER BY F, C, T
8          s_j^i ← fetchRow(S_i)
9          s_j^{i+1} ← fetchRow(S_i)
10         sum_j, sum_j^local ← s_j^i.M
11         count_j, count_j^local ← 1
12     foreach r ∈ R do
13         foreach K_j ∈ X do
14             while ((d(r, s_j^i) ≥ d(r, s_j^{i+1}) ∧ s_j^{i+1}.G = r.G)∨
15             s_j^i.G < r.G) do
16                 if (d(r, s_j^i) > d(r, s_j^{i+1}) ∨ s_j^i.G ≠ r.G) then
17                     sum_j, count_j ← ∅
18                 if (s_j^{i+1}.T ≠ s_j^i.T ∨ r.G > s_j^i.G) then
19                     sum_j^local, count_j^local ← ∅
20                 sum_j, sum_j^local += s_j^{i+1}.M
21                 count_j, count_j^local ++
22                 s_j^i ← s_j^{i+1}
23                 s_j^{i+1} ← fetchRow(S_j)
24             if (r.G = s_j^i.G ∧ d(r, s_j^i) ≤ 365) then
25                 agg_j ← sum_j/count_j
26             sum_j ← sum_j^local
27             count_j ← count_j^local
28         z.add(r.F, r.C, r.T, K', Eval(X, agg_1, ..., agg_l))
29 end
```

The first step, on line 2, is to fetch from the database the formula associated to a derived nutrient $K$'. The formula is expanded as we know from Section 2.3. Then, the nutrients involved in the formula are extracted and saved in an array $X$. On lines 3 and 4 $R$ is initialized selecting $F$, $C$ and $T$ distinctly from the input table $R$ and ordered by $F$, $C$ and $T$. Technically in PL/pgSQL, this query is associated to a cursor. A cursor is similar to a pointer that points

to the query result rows one by one. Then, the referenced row can be fetched from the disk in a local variable, allowing a sequential traversing of all the query result rows [3].

The initialization phase continues for each nutrient $K_j$ inside the array $X$ (lines 5-11). A cursor $S_j$ is initialized selecting $F$, $C$, $T$ and $M$ from the tuples that refer to the nutrient $K_j$, sorted by $F$, $C$ and $T$. At this point two variables are defined, the current value is stored in $s_j^i$ and the next value is stored in $s_j^{i+1}$. The $i$ index corresponds to the position of the cursor inside the table. In the initialization phase, $s_j^i$ points to the first tuple of $S_j$ and $s_j^{i+1}$ points to the second one, for instance $s_2^1 = (Barley, Aarau, 17.02.2007, 2.5)$ and $s_2^2 = (Corn, Bern, 04.08.2011, 3.5)$ in Table 3.1c. After the initialization phase an aggregation set is formed by $sum_j$ and $count_j$ variables, used to aggregate results. This set is initialized with the value of $s_j^i$ and is going to aggregate each nearest neighbor of the current tuple $r$. The same happens with aggregation set $sum_j^{local}$ and $count_j^{local}$, which is going to keep the aggregation in the case the aggregated tuples are also nearest neighbors of the next $r$.

At this point, the algorithm starts to process each tuple of $R$ with the loop of line 12, in a single sequential scan. For every tuple in $R$ it is necessary to find a value of every nutrient in $X$ to be able to evaluate the formula, as we see in the loop on line 13. Now, the idea is to fetch the tuples with minimal distance from $r$ and aggregate them. To do so, we continuously fetch new values into $s_j^{i+1}$ if the distance between $r$ and $s_j^{i+1}$ is smaller than or equal to the distance between $r$ and $s_j^i$ and the groups of $s_j^{i+1}$ and $r$ are the same (see line 14). The group condition is formed by $F||C$, which stands for the name of the feed $F$ concatenated to the name of the canton $C$ and the distance between rows is defined as the temporal distance between timestamps.

A new value in $s_j^{i+1}$ is also fetched in the case that the group of $s_j^i$ is smaller than the group of $r$ (line 15). This means that a new feed or a new canton has been reached in $R$ so that the aggregation must be set to 0 and start again. This system works because of the sorting of $S_j$ by $F$, $C$ and $T$, which ensures that every nearest neighbor is placed one after the other in the cursor, as we have seen in the running example. This means that we can sum the values of the nearest neighbors in the variables $sum_j$ and $sum_j^{local}$ (lines 20-21) for the aggregation until the conditions do not hold anymore.

When $s_j^{i+1}$ reaches the point where it has a bigger distance than $s_j^i$ or a bigger group than $r$, the aggregated average is calculated (lines 24-25). Here comes a particular condition required by the biologist in charge of the Swiss Feed Database: the nearest neighbor(s) used cannot be more distant than 365 days, which means that the distance between $s_j^i$ and $r$ must be shorter than a year in order to be considered a meaningful value (line 24). After that, $sum_j$ takes the value of $sum_j^{local}$ (lines 26-27), to be ready for the next iteration if the aggregated tuples in $sum_j^{local}$ are also nearest neighbors of next $r$. On lines 16-17 and 18-19 we see the conditions necessary to empty the aggregation variables. The $sum_j$ variable is emptied when the distance between r and $s_j^i$ is

bigger than the distance between r and $s_j^{i+1}$ or if the group has changed. The $sum_i^{local}$ variable is emptied when the timestamp of $s_j^{i+1}$ is not equal to the timestamp of $s_j^i$ or the group of $r$ is bigger than the group of $s_j^i$. The $sum_i^{local}$ is only kept for the cases where the aggregated tuples are nearest neighbors also of the next $r$.

Once that a value is loaded into the formula for every nutrient in $X$, the formula is evaluated and added to the results table $Z$ together with feed $F$, canton $C$, timestamp $T$ and derived nutrient $K'$ (line 28).

The essential part of this approach is the initial sorting of the tables $R$ and $S$, which has a complexity of $O(n * log(n))$. The rest of the algorithm is executed with only one sequential scan on the sorted tables, which takes $O(n)$.

# 4 Experimental Evaluation

In this section we are going to experimentally compare the optimized SQL solution ($SQL^+$) with the A-NNJ approach. We are going to present the results of two different experiments: the first one tests the ties management efficiency, the second one reports how the number and the distribution of different $K$ in $S$ changes the performances.

## 4.1 Setup

The experiments were executed on a dual-core 3.33GHz server with 4GB of RAM and a 7200 rpm hard disk.

For all experiments we fixed $R$ as a table having 2 million of rows. $R$ contains always the same group condition $G$ and an increasing timestamp, that goes from 1 to 2 million (see Table 4.1). The group condition $G$ of $R$ has been fixed, because having different $G$ would not affect the computation of derived values for any of the chosen approaches, since $G$ is part of the index on $R$. $S$ also has 2 million of rows with data changing during the experiments as described in the next sections. All the tables used have an index on $G$ and $T$. $K$ has not been included inside the index definition to see how conditions on the nearest neighbor influence performances. The condition we used inside the evaluation process has been $\theta = (K = K^*)$ where $K^*$ is a fixed nutrient, $K^* = 1$.

| id | $G$ | $T$ |
|---|---|---|
| 1 | Corn + Zurich | 1 |
| 2 | Corn + Zurich | 2 |
| 3 | Corn + Zurich | 3 |
| ... | ... | ... |
| 2000000 | Corn + Zurich | 2000000 |

Table 4.1: Table $R$ for the experiments

## 4.2 Ties Management

To investigate the effect of ties in the computation of derived nutrients, we gradually increased the average number of nearest neighbors per row of $R$. In order to do this we built 8 different tables $S$, each of them having 2 million of

rows and having a fixed group condition $G$ and a fixed nutrient $K$. We fixed $G$ and $K$ in order to isolate the problem of ties management, without having any other external effect having different $K$s, while different $G$s would not effect the experiment like explained in Section 4.1, so we kept the setup as simple as possible. The first table (Table 4.1a) contains no ties (in respect to the $R$ described above), the second one (Table 4.1b) has an average of 3 neighbors per row of $R$, this means that, averaged, for each row of $R$, we will have a tie containing 3 measurements, the third one contains an average of 5 neighbors, the fourth one 9, the fifth one 17, the sixth one 33, the seventh one 65 and the last one (Table 4.1c) 129. The timestamps contained in each $S$ always go from 1 to 2 million (with the possibility to have gaps in between), so that for the given $R$ each row of $S$ is the nearest neighbor of at least one row of $R$, i.e. the entire $S$ have to read by both algorithms (same conditions).

(a) $S_{t1}$

| id | $G$ | $T$ | $K$ | $M$ |
|---|---|---|---|---|
| 1 | Corn + Zurich | 1 | 1 | 2.3 |
| 2 | Corn + Zurich | 2 | 1 | 2.6 |
| 3 | Corn + Zurich | 3 | 1 | 1.9 |
| 4 | Corn + Zurich | 4 | 1 | 2.4 |
| ... | ... | ... | ... | ... |
| 2000000 | Corn + Zurich | 2000000 | 1 | 2.2 |

(b) $S_{t2}$

| id | $G$ | $T$ | $K$ | $M$ |
|---|---|---|---|---|
| 1 | Corn + Zurich | 1 | 1 | 1.8 |
| 2 | Corn + Zurich | 1 | 1 | 2.1 |
| 3 | Corn + Zurich | 3 | 1 | 2.0 |
| 4 | Corn + Zurich | 3 | 1 | 1.9 |
| ... | ... | ... | ... | ... |
| 2000000 | Corn + Zurich | 1999999 | 1 | 2.4 |

(c) $S_{t128}$

| id | $G$ | $T$ | $K$ | $M$ |
|---|---|---|---|---|
| 1 | Corn + Zurich | 1 | 1 | 2.2 |
| 2 | Corn + Zurich | 1 | 1 | 2.6 |
| ... | ... | ... | ... | ... |
| 128 | Corn + Zurich | 1 | 1 | 2.4 |
| 129 | Corn + Zurich | 129 | 1 | 2.0 |
| 130 | Corn + Zurich | 129 | 1 | 2.0 |
| ... | ... | ... | ... | ... |
| 2000000 | Corn + Zurich | 1999873 | 1 | 2.1 |

Table 4.2: $S_t$

Figure 4.1 reports the results of these experiments using the A-NNJ and the $SQL^+$ algorithms. On the y-axis the runtime in seconds is reported, on the x-axis the average number of nearest neighbors per row of $R$ is stated.

The runtime for the A-NNJ algorithm is constant during the entire experiment, the major effort is the sorting of both $R$ and $S$, which has a complexity

of $n \cdot log\,(n)$. Since $R$ and $S$ have a constant number of 2 million of rows and since the A-NNJ algorithm only scans both of them once (after having sorted them) its runtime is not affected by the increasing average number of nearest neighbors.

The $SQL^+$ approach shows a linear increasing runtime. The linear increasing trend is given by the fact, that for each row of $R$ an increasing number of tuples have to be read from the disk (since the measurements are not stored in the index), which is very time consuming. When the average number of neighbors is one (no ties at all), for each row of $R$, just one tuple has to be loaded from $S$, but when each row of $R$ has an average of 129 neighbors, then 129 rows of $S$ have to be loaded from the disk for each row of $R$.

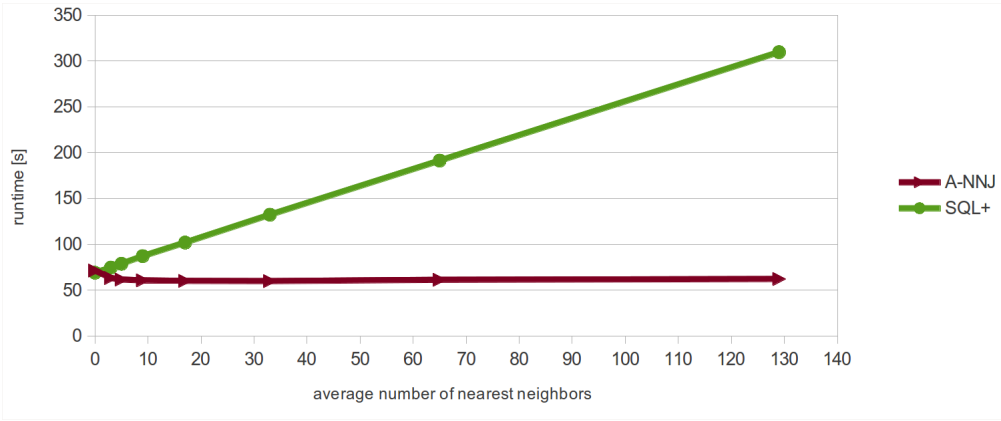



Figure 4.1: Ties management

The only point where the $SQL^+$ approach performs better than the A-NNJ one, is where there are no ties at all (x=1 on the graph). However, in the real case, we know that usually for a given sample more than one measurement is taken for each nutrient, so it will often be the case, that ties are present, which speaks in favor of the A-NNJ approach.

## 4.3 Selectivity of Condition $K = K^*$

In this experiment we want to show the performance of the two algorithms given a nearest neighbor(s) search for a fixed nutrient $K^*$ and a gradually decreasing selectivity of the condition $K = K^*$ on table $S$. The selectivity of $K = K^*$ is given by the number of rows in $S$ storing measurements of $K^*$ divided by the total number of rows in $S$.

**Example:** Given a table $S$ containing 100 rows from which 20 are storing a value for $K^*$, then the selectivity of condition $K = K^*$ is equal to $\frac{\#rows\ containing\ k^*}{total\ number\ of\ rows} = \frac{20}{100} = 20\%$

Similarly to the experiment on ties management we built eight different tables $S$, each of them having two million of rows and having a fixed group condition $G$ as a well as a strictly increasing timestamp, that goes from 1 to 2 million - in order to never have ties and so that the entire $S$ is used. We fixed $G$ because having different $G$s would not effect the experiment (see section 4.1) and so the setup is kept as simple as possible. Fixing $K^* = 1$ as the nutrient for the nearest neighbor(s) search, the eight tables are built like this: each row of the first table (Table 4.2a) contains a measurement of nutrient $K^*$, in this case the selectivity of condition $K = K^*$ is 100%, since the entire $S$ is selected for the search of nearest neighbor(s), in the second one (Table 4.2b) only 50% of the rows contain $K^*$, this means that just half of the rows of $S$ can be candidate nearest neighbors for any row of $R$ (we say, that the selectivity of condition $K = K^*$ decreases), in the third one 25% of the rows contain $K^*$, in the fourth one 12.5%, in the fifth one 6.25%, in the sixth one 3.125%, in the seventh one 1.5625% and in the last one (Table 4.2c) only 0.78125%.

(a) $S_{k1}$

| id | $G$ | $T$ | $K$ | $M$ |
|---|---|---|---|---|
| 1 | Corn + Zurich | 1 | 1 | 2.3 |
| 2 | Corn + Zurich | 2 | 1 | 2.5 |
| 3 | Corn + Zurich | 3 | 1 | 2.0 |
| 4 | Corn + Zurich | 4 | 1 | 1.9 |
| ... | ... | ... | ... | ... |
| 2000000 | Corn + Zurich | 2000000 | 1 | 1.8 |

(b) $S_{k2}$

| id | $G$ | $T$ | $K$ | $M$ |
|---|---|---|---|---|
| 1 | Corn + Zurich | 1 | 1 | 1.8 |
| 2 | Corn + Zurich | 2 | 2 | 4.1 |
| 3 | Corn + Zurich | 3 | 1 | 2.7 |
| 4 | Corn + Zurich | 4 | 2 | 4.2 |
| ... | ... | ... | ... | ... |
| 2000000 | Corn + Zurich | 2000000 | 2 | 4.0 |

(c) $S_{k128}$

| id | $G$ | $T$ | $K$ | $M$ |
|---|---|---|---|---|
| 1 | Corn + Zurich | 1 | 1 | 2.1 |
| 2 | Corn + Zurich | 2 | 2 | 4.5 |
| ... | ... | ... | ... | ... |
| 128 | Corn + Zurich | 128 | 128 | 0.3 |
| 129 | Corn + Zurich | 129 | 1 | 2.4 |
| ... | ... | ... | ... | ... |
| 2000000 | Corn + Zurich | 2000000 | 128 | 0.2 |

Table 4.3: $S_k$

Figure 4.2 reports on the y-axis the runtime in seconds for the A-NNJ and the $SQL^+$ algorithms, on the x-axis we have the selectivity of condition $K = K^*$ in percent. The A-NNJ algorithm shows a linear decreasing runtime as the selectivity of condition $K = K^*$ declines. This is given by the fact, that

when the selectivity decreases, the selection on table $S$ returns less and less tuples, less tuples returned means also less tuples to sort and successively to scan, which are the reasons why the runtime shows a decreasing trend.

The $SQL^+$ approach on the other hand shows an exponential increasing runtime. Remember, that for each row of $R$ the $SQL^+$ query looks for the greatest of all the timestamps in $S$, that is smaller than the timestamp in $R$ and that is of the given nutrient $K^*$. In addition the $SQL^+$ query also searches the smallest of all the timestamps in $S$, that is greater or equal then the timestamp in $R$ and that is of the given nutrient $K^*$. Since the index on $S$ only contains $G$ and $T$, in order to see if a row contains a measurement of the given $K^*$, the tuple must be read from the disk, which is very time consuming. If all the tuples in $S$ store measurements of the given $K^*$ (selectivity of condition $K = K^*$ is 100%), then for the search of the min and the max just described, after an index scan, only two tuples have to be read from the disk, since every tuple in $S$ is of the searched nutrient $K^*$. However, if the selectivity of condition $K = K^*$ is of 50%, then in order to find the min and the max we will have to read from the disk an average of 3 tuples; if the selectivity of condition $K = K^*$ goes down to 25% then the average number of tuples to read from the disk will increase to 9 and so on. Therefore the less the selectivity of condition $K = K^*$, the more the tuples, that we have to read from the disk for the $SQL^+$ approach. The number of tuples to read increases exponentially, this explains the exponential trend for the runtime of the $SQL^+$ approach, like shown in Figure 4.2.

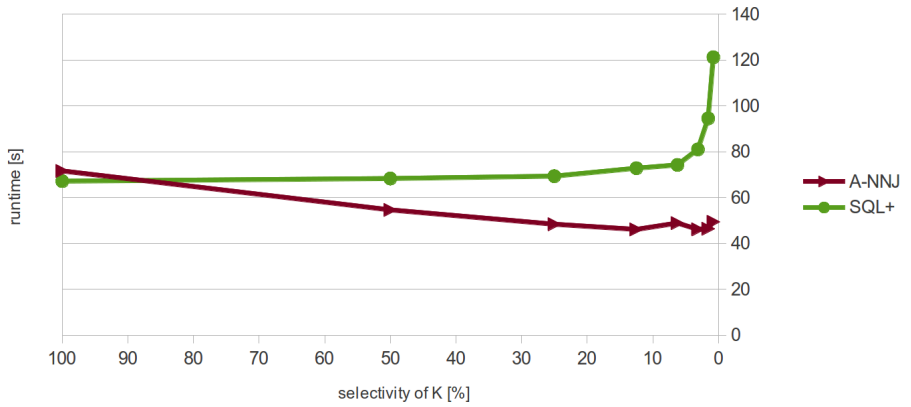



Figure 4.2: Selectivity of K

Also for this experiment there is just one point where the $SQL^+$ approach performs better than the A-NNJ one, precisely when we have a selectivity of condition $K = K^*$ of 100%, i.e. all the rows of $S$ contain measurements of the fixed nutrient $K^*$. However, in the real case, taking the data of the Swiss Feed Database where we have measurements from about 400 different nutrients and

taking $S$ as the entire database, would result in having a selectivity of condition $K = K^*$ for any given $K^*$, that is certainly under 7%, i.e. there is no nutrient in the entire database, which has a selectivity higher than 7%. With this conditions the A-NNJ approach performs much better than the optimized SQL solution.

# 5 Visualization of the Results

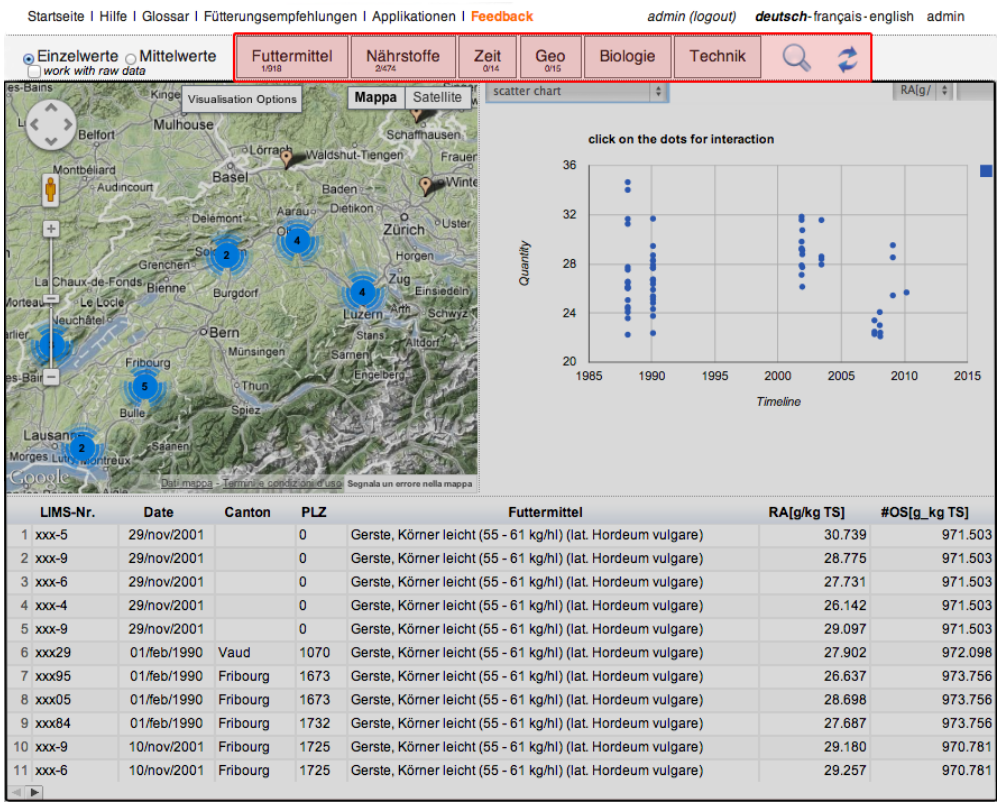
The Swiss Feed Database is a public service designed for companies, farmers or research institutions and collects information about several nutrient parameters of specific feed types. The information is accessible through a web application. The current application only provides manually prepared average data. Therefore Agroscope (Federal Office for Agriculture) and the University of Zurich started a collaboration to design a new web interface and new database techniques to improve the offered functionalities and the quality of the data offered to the users. In particular, the new application permits to display feeds data in specific time frames or with specific biological or geographical parameters.

## 5.1 Online Interface

The new Web Interface Version 2.0 of the Swiss Feed Database has been developed at the University of Zurich. The application is completely Web 2.0 based, which means that it adopts AJAX technology to manage the requests to the server. This is particularly interesting because the user does not need to load the entire web page for each new search request anymore. The chart, the table and the map of Figure 5.1 are built with the help of several JavaScript libraries, especially the Google Chart Tools[1] and the Google Maps API[2]. Our task was to implement the possibility to request information about derived nutrients within the actual application.

The layout of the page consists of two distinct parts: the selection zone in Figure 5.1 (marked in red on the top) serves to select the information a user wants to display like the feed(s) (Futtermittel), the nutrient(s) (Nährstoffe) and the limitations of time (Zeit) and geography (Geo). This way a user can perform very precise queries, getting rich information about the feeds. For the selection of derived nutrients we added a tab in the nutrients (Nährstoffe) menu, as is evidence in Figure 5.2. The second part of the page is composed of the results of the query (marked in gray in Figure 5.1); the results are visualized in a map, a chart and the actual values in the table below. The table shows a column for each nutrient selected, with a hash mark (#) in the abbreviation in the case of a derived nutrient. In addition we also state the feed (Futtermittel), Canton, Zip Code and Date. The measured nutrients also have a LIMS-Number that differentiates the different samples. For the

[1] https://developers.google.com/chart/
[2] https://developers.google.com/maps/




| | LIMS-Nr. | Date | Canton | PLZ | Futtermittel | RA[g/kg TS] | #OS[g_kg TS] |
|---|---|---|---|---|---|---|---|
| 1 | xxx-5 | 29/nov/2001 | | 0 | Gerste, Körner leicht (55 - 61 kg/hl) (lat. Hordeum vulgare) | 30.739 | 971.503 |
| 2 | xxx-9 | 29/nov/2001 | | 0 | Gerste, Körner leicht (55 - 61 kg/hl) (lat. Hordeum vulgare) | 28.775 | 971.503 |
| 3 | xxx-6 | 29/nov/2001 | | 0 | Gerste, Körner leicht (55 - 61 kg/hl) (lat. Hordeum vulgare) | 27.731 | 971.503 |
| 4 | xxx-4 | 29/nov/2001 | | 0 | Gerste, Körner leicht (55 - 61 kg/hl) (lat. Hordeum vulgare) | 26.142 | 971.503 |
| 5 | xxx-9 | 29/nov/2001 | | 0 | Gerste, Körner leicht (55 - 61 kg/hl) (lat. Hordeum vulgare) | 29.097 | 971.503 |
| 6 | xxx29 | 01/feb/1990 | Vaud | 1070 | Gerste, Körner leicht (55 - 61 kg/hl) (lat. Hordeum vulgare) | 27.902 | 972.098 |
| 7 | xxx95 | 01/feb/1990 | Fribourg | 1673 | Gerste, Körner leicht (55 - 61 kg/hl) (lat. Hordeum vulgare) | 26.637 | 973.756 |
| 8 | xxx05 | 01/feb/1990 | Fribourg | 1673 | Gerste, Körner leicht (55 - 61 kg/hl) (lat. Hordeum vulgare) | 28.698 | 973.756 |
| 9 | xxx84 | 01/feb/1990 | Fribourg | 1732 | Gerste, Körner leicht (55 - 61 kg/hl) (lat. Hordeum vulgare) | 27.687 | 973.756 |
| 10 | xxx-9 | 10/nov/2001 | Fribourg | 1725 | Gerste, Körner leicht (55 - 61 kg/hl) (lat. Hordeum vulgare) | 29.180 | 970.781 |
| 11 | xxx-6 | 10/nov/2001 | Fribourg | 1725 | Gerste, Körner leicht (55 - 61 kg/hl) (lat. Hordeum vulgare) | 29.257 | 970.781 |

Figure 5.1: Online interface

derived nutrients this information is not available, because the nutrients used to evaluate the formula usually come from different samples.

During the development, the system encountered a big performance problem in the visualization of the results when the query was returning a big amount of data. The local JavaScript engine in the user's browser quickly shows its limitations when it comes to process and visualize MBs of information, especially on Microsoft Internet Explorer. To solve this problem, the developers decided to limit the dataset of the query to 150 samples by default. This value can be increased in the Web Interface if the browser supports a higher quantity of data. For the A-NNJ function this means that the input $R$ is limited to 150 entries by default. Working with less samples could influence the quality of the computed derived nutrients values.

## 5.2 PHP implementation

The application is written in PHP on the server side for the interaction with the database. This means in most cases that PHP gets the data from the database, formats them as JSON objects and returns them to the browser for the JavaScript visualization. The implementation of the functionality to compute derived nutrients values was possible changing only a few PHP files;

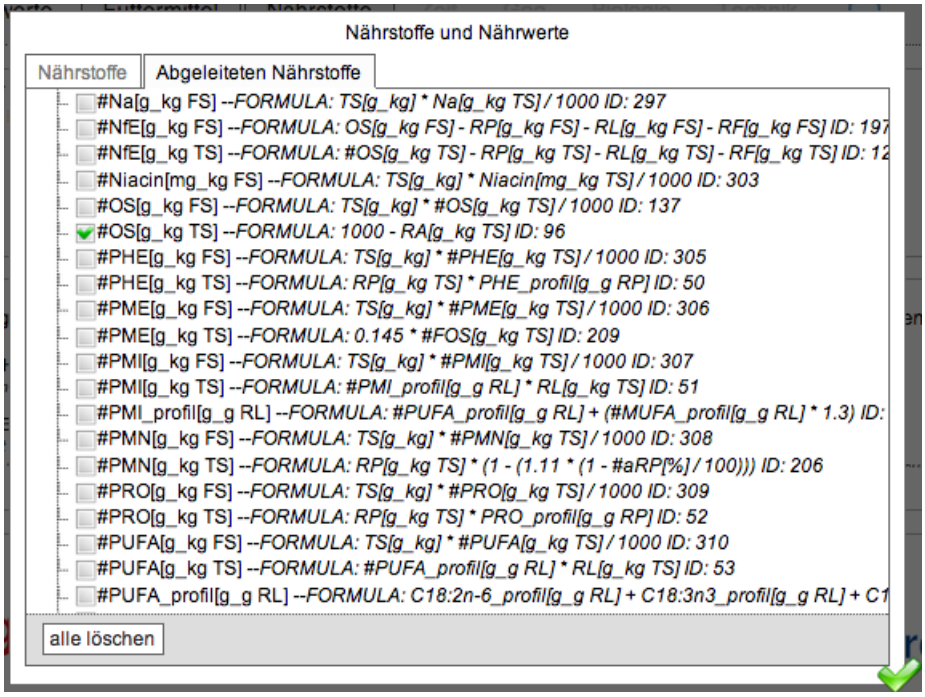



Figure 5.2: The derived nutrients tab

for the scope of this report, we are going to describe only the application parts affected by our implementation.

The selection of feeds, nutrients, temporal and geographical restrictions is mapped to an AJAX call to the server to get the information. The calls sequence is shown in the diagram of Figure 5.3. The feeds in the feed menu are read from static files to have a faster initial page loading. Once some feeds are selected in the menu, a call to the server starts interrogating the database to get nutrients and derived nutrients associated to the selected feeds. This happens inside the *get_td_nutrients.php* and *get_td_nutrients_derived.php*. The measured nutrients returned are only those that have currently available measurements in the database for the selected feeds. The derived nutrients menu displays every entry that makes sense for the given feeds. The mapping between feeds and derived nutrients that make sense for these feeds is stored in a table named *t_formula_feed* in the database. The *get_td_nutrients_derived.php* joins table *t_formula* and table *t_formula_feed* limiting the results to the selected feeds. This data are then returned to the browser and displayed inside the nutrients menu.

The selections of nutrients and derived nutrients inside the menu trigger a request for time and geographical limitations. These menus display the limitations available for the feeds and nutrients selected. The limitations of time are structured in years and seasons, and the limitations of geography in canton, altitude and radius, e.g. 2 km around Zurich (see Figure 5.4). These limitations menus allow the user to get very specific and personalized

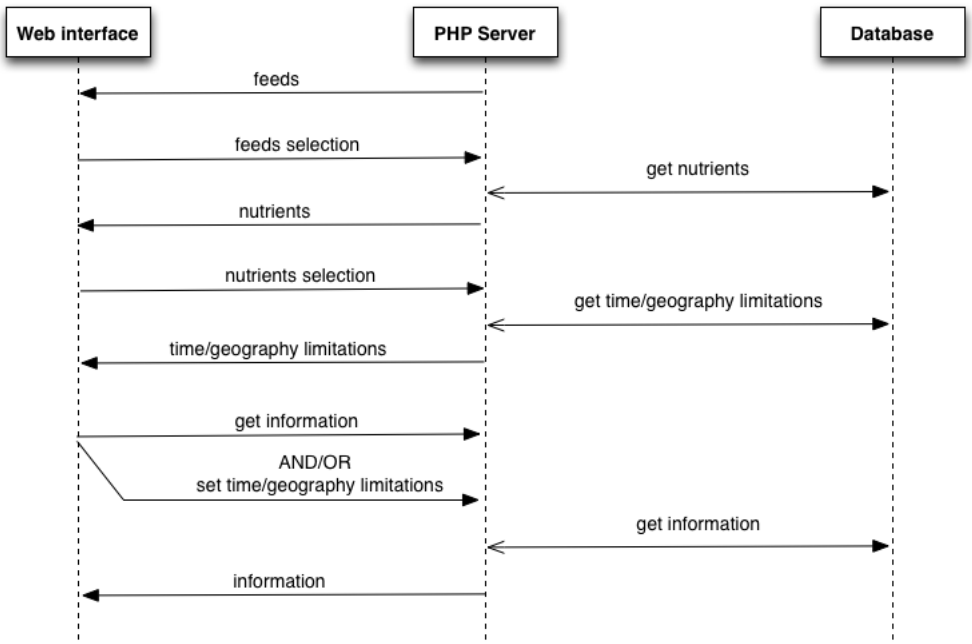



Figure 5.3: Sequence of calls to the PHP server

information about feeds in a region or a specific season.

Clicking on the magnifier on the right top of Figure 5.1 the query is executed. This event triggers an AJAX-call to the server to get the information desired, based on the selections in the menus, and display them in the web interface. All this happens inside the *ajax-samples.php* file, which collects the nutrients values from the database and formats them in a JSON object for the front-end visualization.

The central part of *ajax-samples.php* is the object called *FeedSample*. Inside it, every nutrient is initialized in the arrays for the nutrients and derived nutrients, setting a value for the abbreviation. It also contains all the variables needed for the visualization of the results, like the *feed name*, *LIMS-number*, *canton*, *date*, *postal code*, the coordinates for the map and the *measurement*. These variables depend on the rows of the response from the nutrients and derived nutrients query. The object also contains a function to reset the nutrient measurements and, most important, the *print_to_json* and *print_to_excel* functions. As the name suggests, *print_to_json* fills a JSON object with the right format for the JavaScript visualization tools and returns it to the browser. The *print_to_excel* is a functionality for subscribed users to download the table of the request results in form of an excel sheet.

The execution of *ajax_samples.php* begins with the creation of the *FeedSample* object. After that, we need to define the database queries to get the information about nutrients and derived nutrients. A query is defined to get 150 randomly chosen samples from all the available samples. The random

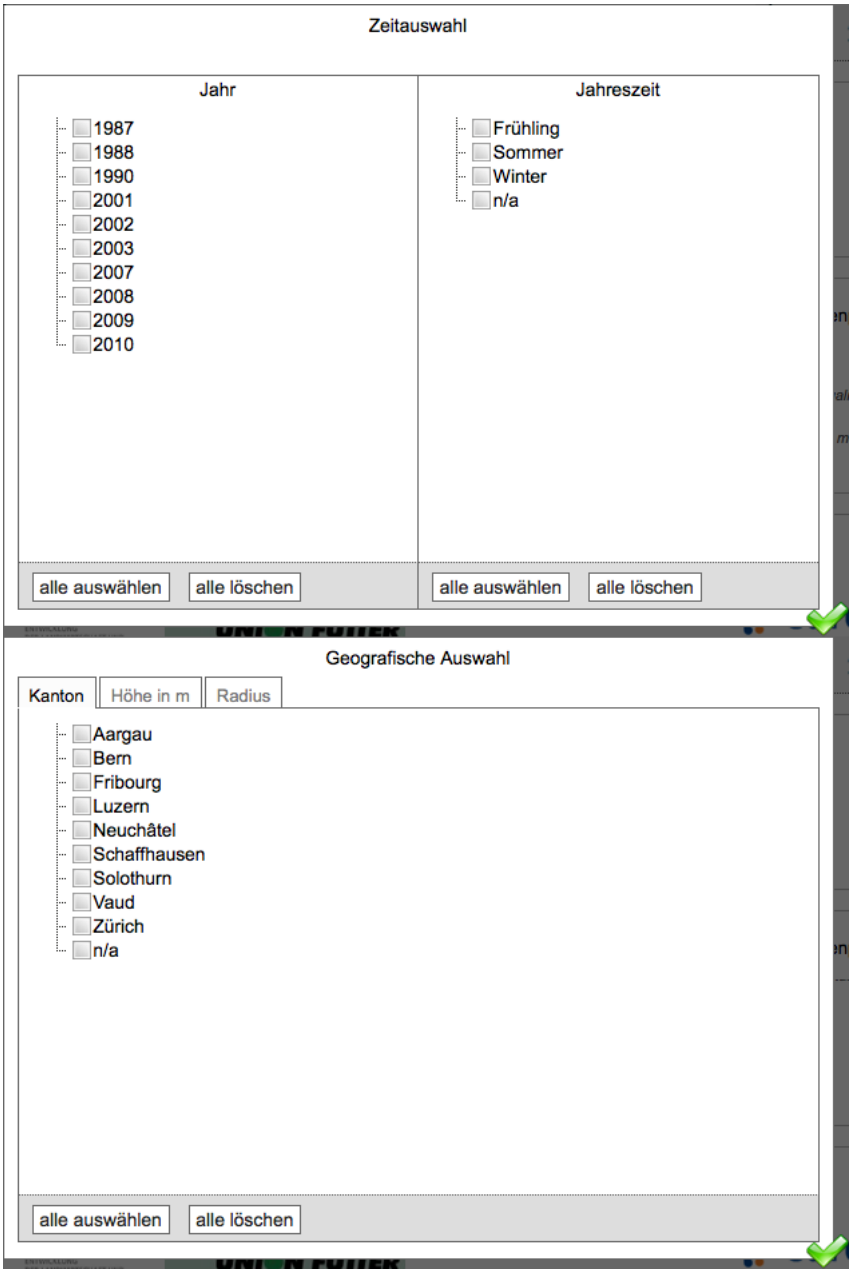



Figure 5.4: The time and geo menus

function is driven in a way that for the same session, always the same "random" samples are chosen, using the setseed(0.123) setting. This is necessary, because the query for measured nutrients is different from the query for derived nutrients, but the samples used must be the same to have the same samples visualized in the web interface.

After this, one query for the nutrients and one for the derived nutrients are defined, using the query for the 150 samples inside an *IN ()* construct to limit the results. The queries are executed inside the DBMS and the results are saved in the variables *nutrients_results* and *derived_nutrients_results*. Then, a loop goes through all the result rows to print them inside the JSON object or the excel sheet. To do so, the function *readNextSampleFromTheQueryResult()* is called. This function reads the next row in the *nutrients_results* and sets every variable inside the *FeedSample* object for the new row. After this the *print_to_json* or *print_to_excel* function is called and the loop goes to the next row. If derived nutrients are selected too, the *readNextSampleFromTheQueryResult()* function searches for the nutrient row a derived nutrient row with the same characteristics inside the *derived_nutrients_results*; if one is found, also the value of the derived nutrient is set inside the *FeedSample* object, otherwise it remains blank. The JSON object is then used by the JavaScript Library to visualize the information in the browser. The excel sheet, on the other hand, is prepared server-side using the *PHPExcel* Library[3] and is returned as a download.

One important aspect during the development was to avoid unnecessary communications with the database server. Therefore *ajax-samples.php* groups the queries in one query for measured nutrients and one for derived nutrients.

The extensive use of AJAX and JavaScript technology leads to an interactive and pleasant user experience. The application is developed with the procedural programming paradigm and we find an object only in the *feed_sample.php*. A complete Object Oriented design would be preferable, because it would permit an easier understanding of the code by new developers and an easier implementation of new functionalities; however, this would mean a complete redesign of the entire application.

[3] http://phpexcel.codeplex.com/

# 6 Conclusions

The achievements of this project can be summarized in two points. First of all, we found two approaches to compute derived nutrients values: an optimized SQL solution ($SQL^+$) and the A-NNJ algorithm in the form of a PL/pgSQL function. Among the two approaches we empirically proved - with the help of two different experiments - that the A-NNJ algorithm almost always performs better than the $SQL^+$ approach. In the ties management experiment, the A-NNJ algorithm shows a constant runtime, while the $SQL^+$ approach has a linear increasing one. Reducing the selectivity of condition $K = K^*$ lets the runtime for the A-NNJ algorithm sink, while the $SQL^+$ approach shows an exponential increasing runtime. The only case where the $SQL^+$ approach would be better, is when there are no ties or when the selectivity of condition $K = K^*$ is equal to 100%. However, we have seen, that in the real case, when the data is taken from the Swiss Feed Database, the dataset results in having a small selectivity (less than 7%) for each possible $K^*$ as well as in having ties in the majority of the cases; these conditions clearly speak in favor of the A-NNJ algorithm.

As a second achievement, the integration of the results into the online interface of the Swiss Feed Database permits a fast and user-friendly visualization of the computed derived nutrients values.

# Bibliography

[1] Christian Böhm, Florian Krebs: The k-nearest neighbour Join: Turbo Charging the KDD Process. Knowledge and Information Systems. (2004)

[2] Byn Yao, Feifei Li, Piyush Kumar: K Nearest Neighbor Queries and KNN-Joins in Large Relational Databases (Almost) for Free. Data Engineering (ICDE) Conference. (2010)

[3] PostgreSQL: Cursors. http://www.postgresql.org/docs/9.2/static/plpgsql-cursors.html. (december 2012)