Vertiefung - Report

Linear Optimization in Relational Databases

University of Zürich - Database Technology Group

Author:
Pascal Zehnder

Advisor:
Georgios Garmpis

Chair of:
Prof. Dr. Michael Böhlen

Universität Zürich[UZH]

October 31, 2016

Contents

1 Introduction

The task for this work was to get acquaint with the linear optimization problem and the most famous algorithm experienced in this area, called Simplex and then implement this algorithm in the context of relational databases as a set of user defined functions in PostgreSQL. For the purpose of in-database analytic methods the open-source library MADlib was used.
An optimization problem has the goal to maximize or minimize a value corresponding to an objective function. The linear optimization problem is a specific optimization problem characterized by a linear objective function and constraints that are expressed as a set of linear equalities or inequalities. Moreover, the diet optimization problem is a subset of the linear optimization problems. The standard form of this problem is a minimization of the objective function with constraints representing an upper bound. For example this problem may be used by a dietitian of an army, who needs to find the most economical diet satisfying the basic minimum nutritional requirements (constraints) for a good food balance (Luenberger & Ye, 2016a). The most popular algorithm for solving this problem is Simplex. The idea of Simplex is to get iteratively from one feasible solution to another one that finally results in a better value of the objective function. This needs to be done until an optimum (minimum or maximum) is reached or the problem gets declared as unbounded (Luenberger & Ye, 2016b).

2 Simplex Algorithm

A linear program is a subset of an optimization problem (maximum or minimum) where the objective function is linear and the constraints consist either of linear equalities or inequalities (Luenberger & Ye, 2016b). The following form represents the standard form of a linear problem:

$$\begin{aligned} \text{minimize } & z = c_1x_1 + c_2x_2 + ... + c_nx_n \\ \text{subject to: } & a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = b_1 \\ & a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n = b_2 \\ & ... \\ & a_{m1}x_1 + a_{m2}x_2 + ... + a_{mn}x_n = b_m \\ & and\ x_1 \geq 0, x_2 \geq 0, ..., x_n \geq 0 \end{aligned}$$

In matrix notation this is shown as below:

$$\begin{aligned} \text{minimize } & z = c^T x \\ \text{subject to: } & Ax = b \\ & and\ x \geq 0 \end{aligned}$$

The corresponding dual problem of this linear problem can be achieved by transforming some number of vectors and matrices. This leads to the following matrix notation:

$$\begin{aligned} \text{maximize } & z = b^T y \\ \text{subject to: } & A^T y = c \\ & and\ xy \geq 0 \end{aligned}$$

The relationship between the primal and the dual form of a linear problem depends on finding a solution. If the primal has an optimal solution then the dual has either. If the primal problem is unbounded then the dual is infeasible, and if the solution of the primal is infeasible then the corresponding dual has an unbounded solution.

Slack and surplus variables Usually linear programming problems naturally arise in inequalities. To convert those problems into the standard form shown above, an inequality may be converted to an equality using slack or surplus variables.

$$a_{11}x_1 + a_{12}x_2 + ... + a_{1n}xn \leq b_1$$

leads to,

$$a_{11}x_1 + a_{12}x_2 + ... + a_{1n}xn + s_1 = b_1, \text{ with } s_1 \geq 0$$

Using the slack variables $s_i, \forall i \in \mathbb{R}$ for m constraints (rows), m columns got added to the matrix, which results in an identity matrix of size $m \times m$.

$$I = \begin{bmatrix} 1 & 0 & ... & 0 & 0 \\ 0 & 1 & ... & 0 & 0 \\ ... & ... & ... & ... & ... \\ 0 & 0 & ... & 1 & 0 \\ 0 & 0 & ... & 0 & 1 \end{bmatrix}$$

Figure 1: Identity matrix

On the other side, surplus variables $l_i, \forall i \in \mathbb{R}$ would be used in case of an inequality that is bigger than some value. Those variables would then be subtracted from this inequality.

$$a_{11}x_1 + a_{12}x_2 + ... + a_{1n}xn \geq b_1$$

leads to,

$$a_{11}x_1 + a_{12}x_2 + ... + a_{1n}xn - l_1 = b_1, \text{ with } l_1 \geq 0$$

The mixing of slack and surplus variables and the existence of only surplus variable wouldn't lead to an identity matrix as shown in Figure 1.

Example In this section the simplex algorithm is demonstrated in its standard form, as well as in its revised form. Throughout this paper, the following linear program is used. This problem is the dual of a diet optimization problem, because it is maximizing the objective function and has only upper bound constraints. It is used for this example to avoid the two-phase method where we would have to add *artifical variables*. The two-phase method consists of phase I where artificial variables are introduced and the sum of artificial variables is the objective function so that a basic feasible solution is found. The phase II then uses the computed solution from phase I and minimizes the original objective function (Luenberger & Ye, 2016b).

$$\begin{aligned}
\text{maximize } z &= 7x_1 + 9x_2 + 4x_3 \\
\text{subject to: } 2x_1 + 1x_2 + 4x_3 &\leq 6 \\
2x_1 + 3x_2 + 2x_3 &\leq 13 \\
4x_1 + 1x_2 + 4x_3 &\leq 12 \\
2x_1 + 5x_2 + 1x_3 &\leq 10 \\
x_1, x_2, x_3 &\geq 0
\end{aligned}$$

The first step in solving this optimization problem is to transform the inequalities into equalities. For this purpose we can define additional variables, so called *slack variables*. This leads to the following *standard form* of an optimization problem:

$$\begin{aligned}
\text{maximize } z &= 7x_1 + 9x_2 + 4x_3 \\
\text{subject to: } 2x_1 + 1x_2 + 4x_3 + s_1 &= 6 \\
2x_1 + 3x_2 + 2x_3 + s_2 &= 13 \\
4x_1 + 1x_2 + 4x_3 + s_3 &= 12 \\
2x_1 + 5x_2 + 1x_3 + s_4 &= 10 \\
x_1, x_2, x_3, s_1, s_2, s_3, s_4 &\geq 0
\end{aligned}$$

2.1 Simplex Algorithm

The goal of the simplex algorithm, and especially for the diet optimization problem, is to iteratively decrease the value of an objective function (minimize). The target is to get from one feasible solution to another feasible solution. As a result, Simplex returns values of the unknown variables and the corresponding total minimized cost of the optimization problem.

2.1.1 Basic and Non-Basic variables

Next, we need to introduce the basic and non-basic variables. All basic variables have assigned a value greater or equal to zero. At the beginning of the algorithm the basic variables are the slack variables. At contrast the non-basic variables always have value zero. After every iteration a basic variable is exchanged to a non basic variable and vice versa. For this purpose we need to split the matrices and vectors as shown below:

$$A = \begin{bmatrix} D & B \end{bmatrix}, c = \begin{bmatrix} c_D \\ c_B \end{bmatrix}, x = \begin{bmatrix} x_D \\ x_B \end{bmatrix},$$

where x_D are the non-basic variables and x_B are the basic variables. Matrix A is divided into two matrices D containing non-basic variables and B containing basic variables. At the first iteration B represents the identity matrix shown in Figure 1.

Initial Tableaux The linear problem from above must be transformed as described into a system containing only of equalities. This can be achieved using slack variables for each inequality. Usually the simplex algorithm is given implemented for minimization. In case that maximizing is the goal of the optimization problem, then the coefficients of the objective values must be

multiplied by -1. At the beginning of the simplex procedure we assume to start with a basic feasible solution and a tableaux corresponding to the following construct that represents the system of equalities:

x_D^T	x_B^T	RHS	basis
D	B	b	x_B
$-c_D^T$	$-c_B^T$	0	$-z$

Table 1: Simplex Initial Tableaux

where RHS defines the right hand side of the equations. For the provided example following values can be assigned to the matrices and vectors to build up the initial tableaux:

$$D = \begin{bmatrix} 2 & 1 & 4 \\ 2 & 3 & 2 \\ 4 & 1 & 4 \\ 2 & 5 & 1 \end{bmatrix}, B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, b = \begin{bmatrix} 6 \\ 13 \\ 12 \\ 10 \end{bmatrix},$$

$$c_D^T = \begin{bmatrix} 7 & 9 & 10 \end{bmatrix}, c_B^T = \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}, x_B = \begin{bmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \end{bmatrix}, z = 0$$

2.1.2 Pivot operation

The pivot operation then generates a new system of equalities based on the old one by replacing a basic variable with a non-basic variable and the other way around (=swapping). In each iteration the table (matrix) is transformed using standard elementary operations (scalar row multiplication, add and subtract rows) (Luenberger & Ye, 2016b).

2.1.3 Example

The initial tableaux (standard form in Table 1) of the example described with its values filled in is shown in Table 2 below:

x_1	x_2	x_3	s_1	s_2	s_3	s_4	b	ratio	basis
2	1	4	1	0	0	0	6	6	s_1
2	3	2	0	1	0	0	13	13/3	s_2
4	1	4	0	0	1	0	12	12	s_3
2	⑤	1	0	0	0	1	10	2	s_4
-7	-9	-4	0	0	0	0	0		$-z$

Table 2: Initial Tableaux

After creating the initial tableaux the basic feasible solution can be detected from the tableaux. The non basic variables x_1, x_2 and x_3 have assigned value zero. The slack variables correspond to the basic variables which means that following solution results as the basic feasible solution:

$$x_1 = 0, x_2 = 0, x_3 = 0 \text{ and } z = 0$$

For every iteration the simplex algorithm contains of the following steps:

1. **Check optimality:** The relative cost coefficients are shown in the last row of the tableaux. If all of the coefficients are non-negative (positive), then the current solution is already the optimal solution, because when having negative values there is still room for improvement of the feasible solution.
   In the provided example there are 3 negative coefficients which means the solution is not optimal yet.

2. **Find entering variable:** If improvements are possible (negative values), a variable needs to be selected as the entering variable. If there exists more than one negative relative cost coefficient it is common practice to choose the most negative value to achieve the greatest decrease of the objective function. The chosen variable then enters the basis while increasing the value of this variable to a positive value (≥ 0), and the corresponding column is called the pivot column.
   For the initial tableaux the entering variable is x_2 which means that $q = 1$ (second element of pivot column).

3. **Check boundness and find leaving variable:** The row with the smallest non negative ratio of b (RHS) divided by the pivot column is chosen as the pivot row, justified by the need of satisfying all constraints. If the values are all negative then the solution is to be considered as unbounded. If there are ratios that are tie, any minimum ratio element can be used as a pivot element.
   The ratios in this example can be gathered from the initial tableaux. The lowest (non-negative) ratio is 2 in the 4th row. Therefore $k = 3$ and the variable s_4 is considered as leaving variable of the basis.

4. **Update tableux:** As last step, the tableaux is updated to an equivalent system of equalities that represent the new solution. To achieve this new solution, a set of elementary row operations (Gaussian elimination) are applied to transform the pivot column into a column of the identity matrix. The operations that must be applied for this step are:
   a) Divide the pivot row (fourth row) by 5
   b) First row minus 1 times new pivot row
   c) Second row minus 3 times new pivot row
   d) Third row minus 1 times new pivot row
   e) Fifth row plus 9 times new pivot row

This leads to following tableaux after the 1st iteration:

x_1	x_2	x_3	s_1	s_2	s_3	s_4	b	ratio	basis
1.6	0	3.8	1	0	0	-0.2	4	2.5	s_1
0.8	0	1.4	0	1	0	-0.6	7	8.75	s_2
3.6	0	3.8	0	0	1	-0.2	10	25/9	s_3
0.4	1	0.2	0	0	0	0.2	2	5	x_2
-3.4	0	-2.2	0	0	0	1.8	18		$-z$

Table 3: Tableaux after 1st iteration

Because there are still some negative relative costs, the algorithm needs to continue with a further iteration:

1. **Check optimality:** The current solution can be improved by updating the tableaux.
2. **Find entering variable:** x_1 has to be chosen as the entering variable.
3. **Check boundness and find leaving variable:** The ratios per row are shown in Table 3. The smallest value can be found in the first row, which leads to the pivot element marked in the circle.
4. **Update tableux:** For updating the tableaux following operations need to be executed:
   a) Divide the pivot row (first row) by 1.6
   b) Second row minus 0.8 times new pivot row
   c) Third row minus 3.6 times new pivot row
   d) Fourth row minus 0.4 times new pivot row
   e) Fifth row plus 3.4 times new pivot row

The following tableaux results after finishing 2 iterations:

x_1	x_2	x_3	s_1	s_2	s_3	s_4	b	basis
1	0	2.375	0.625	0	0	−0.125	2.5	x_1
0	0	−0.5	−0.5	1	0	−0.5	5	s_2
0	0	−4.75	−2.25	0	1	0.25	1	s_3
0	1	−0.75	−0.25	0	0	0.25	1	x_2
0	0	5.875	2.125	0	0	1.375	26.5	$-z$

Table 4: Tableaux after 2nd iteration

Finally there are no negative values in c_b anymore, meaning that the optimal solution is found in the beginning of the 3rd iteration:

$$z = -26.5, x_1 = 2.5, x_2 = 1, x_3 = 0.$$

2.2 Revised Simplex Algorithm

The revised simplex algorithm is a modification of the simplex algorithm. It is following the same steps as Simplex, but it doesn't use the large matrix A for representing the current state after every iteration. Instead of matrix A it is using B^{-1} which is computed after every iteration based on the previous B^{-1} and the original A, which means that matrix A will never be changed throughout the whole algorithm. Based on matrix B^{-1} everything can be fully computed using the originals of A, b, c.

Steps of the algorithm

1. Initialize matrices and vectors:
   $z = 0, y^T = c_B^T B^{-1}, \beta = B^{-1} b$
2. Iteration:

a) **Step 1 (Compute Relative Costs):** Calculate the current relative cost coefficients $r_D^T = c_D^T - y^T D$ where $y^T = c_B^T B^{-1}$. D is the part of matrix A where the columns correspond to the initialized non-basis variables and the same for c_D^T, which is part of vector c. If $r_D \geq 0$ then stop, the current solution is optimal. Additionally, find the entering variable q by selecting the smallest relative cost coefficient.

b) **Step 2 (Compute Pivot Column):** Calculating $\overline{a}_q = B^{-1}a_q$ returns the vector for using as the pivot column where a_q is the pivot column of A.

c) **Step 3 (Determine Leaving Variable):** If no value of $\overline{a}_q > 0$ then stop, the problem is unbounded. Else, compute the ratios $\beta/\overline{a}_q$ to determine the pivot element where $\beta = B^{-1}b$.

d) **Step 4 (Update Tableaux):** The pivot column now needs to enter the basis array which means updating B^{-1} and other vectors and matrices. The following operations must be executed in correct order:

i. $B^{-1}[k, *] = \frac{B^{-1}[k,*]}{\overline{a}_q}$ (divide pivot row by the corresponding $\overline{a}_q$)

ii. $B^{-1}[i, *] - \frac{B^{-1}[k,*]}{\overline{a}_q}, \forall i \neq k$ (subtract from every row the new calculated pivot row)

iii. $swap(c_D[q], c_B[k])$ and swap the position in `basis` and `nonbasis` array.

iv. $\beta = B^{-1}b$

v. $y^T = c_B^T B^{-1}$

vi. $z = c_B^T \beta$

Example According to the example used in this paper, the algorithm is executed as follows:

1. Initialize the matrices and vectors:

$$B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad D = \begin{bmatrix} 2 & 1 & 4 \\ 2 & 3 & 2 \\ 4 & 1 & 4 \\ 2 & 5 & 1 \end{bmatrix}$$

$$c_B^T = \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}$$

$$c_D^T = \begin{bmatrix} -7 & -9 & -4 \end{bmatrix}$$

$$y_D^T = \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\beta = \begin{bmatrix} 6 \\ 13 \\ 12 \\ 10 \end{bmatrix}$$

2. First Iteration

a) **Step 1 (Compute Relative Costs):**

$$r_D^T = c_D^T - y^T D = \begin{bmatrix} -7 & -9 & -4 \end{bmatrix}$$

additionally set $q = 1$ (second element of the vector)

b) **Step 2 (Compute Pivot Column):**

$$\overline{a}_q = B^{-1}a_q = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 3 \\ 1 \\ 5 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ 1 \\ 5 \end{bmatrix}$$

c) **Step 3 (Determine Leaving Variable):**

$$\text{ratio} = \beta/\overline{a}_q = \begin{bmatrix} 6 \\ 13/3 \\ 12 \\ 2 \end{bmatrix}$$

therefore the last value is the minimum ratio and $r = 3$.

d) **Step 4 (Update Tableaux):** Updating B^{-1} according to column $\overline{a}_q$

$$B^{-1} = \begin{bmatrix} 1 & 0 & 0 & -0.2 \\ 0 & 1 & 0 & -0.6 \\ 0 & 0 & 1 & -0.2 \\ 0 & 0 & 0 & 0.2 \end{bmatrix}$$

$$\text{new } c_B^T = \begin{bmatrix} 0 & 0 & 0 & -9 \end{bmatrix}$$

$$\text{new } c_D^T = \begin{bmatrix} -7 & 0 & -4 \end{bmatrix}$$

$$\beta = B^{-1}b = \begin{bmatrix} 4 \\ 7 \\ 10 \\ 2 \end{bmatrix}$$

$$y^T = c_B^T B^{-1} = \begin{bmatrix} 0 & 0 & 0 & -1.8 \end{bmatrix} \qquad z = c_B^T \beta = -18$$

3. Second iteration

a) **Step 1 (Compute Relative Costs):**

$$r_D^T = c_D^T - y^T D = \begin{bmatrix} -7 & 0 & -4 \end{bmatrix} - \begin{bmatrix} 0 & 0 & 0 & -1.8 \end{bmatrix} \begin{bmatrix} 2 & 1 & 4 \\ 2 & 3 & 2 \\ 4 & 1 & 4 \\ 2 & 5 & 1 \end{bmatrix} = \begin{bmatrix} -3.4 & 0 & -2.2 \end{bmatrix}$$

additionally set $q = 0$ (first element of the vector)

b) **Step 2 (Compute Pivot Column):**

$$\overline{a}_q = B^{-1}a_q = \begin{bmatrix} 1 & 0 & 0 & -0.2 \\ 0 & 1 & 0 & -0.6 \\ 0 & 0 & 1 & -0.2 \\ 0 & 0 & 0 & 0.2 \end{bmatrix} \begin{bmatrix} 2 \\ 2 \\ 4 \\ 2 \end{bmatrix} = \begin{bmatrix} 1.6 \\ 0.8 \\ 3.4 \\ 0.4 \end{bmatrix}$$

c) **Step 3 (Determine Leaving Variable):**

$$\text{ratio} = \beta/\overline{a}_q = \begin{bmatrix} 2.5 \\ 8.75 \\ 50/17 \\ 5 \end{bmatrix}$$

therefore the first value is the minimum ratio and $r = 0$.

d) **Step 4 (Update Tableaux):** Updating B^{-1} according to column $\overline{a}_q$

$$B^{-1} = \begin{bmatrix} 0.625 & 0 & 0 & -0.2 \\ -0.5 & 1 & 0 & -0.6 \\ -2.25 & 0 & 1 & -0.2 \\ -0.25 & 0 & 0 & 0.2 \end{bmatrix} \qquad \text{new } c_B^T = \begin{bmatrix} -7 & 0 & 0 & -9 \end{bmatrix} \qquad \beta = B^{-1}b = \begin{bmatrix} 2.5 \\ 5 \\ 1 \\ 1 \end{bmatrix}$$

$$\text{new } c_D^T = \begin{bmatrix} 0 & 0 & -4 \end{bmatrix}$$

$$y^T = c_B^T B^{-1} = \begin{bmatrix} -2.125 & 0 & 0 & -1.375 \end{bmatrix} \qquad z = c_B^T \beta = -26.5$$

4. Third iteration:

a) **Step 1 (Compute Relative Costs):**

$$r_D^T = c_D^T - y^T D = \begin{bmatrix} 0 & 0 & -4 \end{bmatrix} - \begin{bmatrix} -2.125 & 0 & 0 & -1.375 \end{bmatrix} \begin{bmatrix} 2 & 1 & 4 \\ 2 & 3 & 2 \\ 4 & 1 & 4 \\ 2 & 5 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 5.875 \end{bmatrix}$$

b) As there is no negative value, we've found the optimal solution:

$$z = -26.5, x_1 = 2.5, x_2 = 1, x_3 = 0$$

.

2.3 Comparison of Simplex and Revised Simplex Algorithm

The standard simplex algorithm updates the entire tableaux after each iteration. As not all values in the tableaux are needed in every iteration, the standard simplex algorithm does computations that are not really needed. In contrast, the revised simplex does not keep a representation of the full tableaux and should therefore be more efficient. This is shown in following computation of time complexity for the theoretical algorithm:

Standard Simplex The advantage of the standard simplex algorithm is its simplicity. A disadvantage of the algorithm is that it updates the complete A-matrix after every iteration, which means it uses $m \times n$ operations to update the complete matrix.

- Computation: $O(mn)$ operations for each iteration

Revised Simplex The revised simplex algorithm needs the following 4 operations with its corresponding upper bound computation time:

Operation	**Sub-Operation**	**Cost**	**Total Cost**
Compute Relative Costs			$O(m^2 + mn)$
	Multiplication of $y^T = c_B^T B^{-1}$	$O(m^2)$	
	Multiplication of $y^T D$	$O(mn)$	
	Subtraction of $r_D = c_D^T - y^T D$	$O(n)$	
Compute Pivot Column	Compute $\overline{a}_q = B^{-1} a_q$	$O(m^2)$	$O(m^2)$
Determine Leaving Variable	Compute ratios $\beta / \overline{a}_q$	$O(m)$	$O(m)$
Update Tableaux			$O(m^2 + mn)$
	$B^{-1}[k, *] = \frac{B^{-1}[k,*]}{\overline{a}_q}$	$O(m)$	
	$B^{-1}[i, *] - \frac{B^{-1}[i,*]}{\overline{a}_q} \forall i \neq k$	$O(m^2)$	
	$swap(c_D[q], c_B[k])$	$O(1)$	
	$\beta = B^{-1} b$	$O(m^2)$	
	$y^T = C_B^T B^{-1}$	$O(m^2)$	
	$z = c_B^T \beta$	$O(m)$	

Leading to following total upper bound for the computation of the revised simplex algorithm:

- Computation: $O(m^2 + mn)$ operations for each iteration

This actually contradicts the efficiency of the revised simplex algorithm. Usually in practice $m << n$ and matrix A is a sparse matrix, meaning to have very few nonzero values. We define τ as the number of nonzero values in matrix A. Since the revised simplex does not update A, it preserves its sparsity and takes advantage of it.

Inserting this τ and assuming that $n >> m$ gives following calculations for the upper bound of the revised simplex:

$$O(m^2 + mn) = O(m^2 + \tau)$$

Compared to the standard simplex, that fills matrix A in with non-zero values (after some iterations), this means for $n >> m$ that the revised simplex is much more efficient for matrices with many nonzero values. Therefore this algorithm was used for the implementation.

3 MADlib

MADlib is an open-source library of in-database scalable analytic methods that can be executed in a relational database engine supporting extensible SQL. MADlib was developed by people from industry (Greenplum) and academia (universities of Berkley, Florida and Wisconsin) in 2008 with the primary target of accelerating data analytics algorithm implementation on DBMS (Database Management System) (Hellerstein *et al.* , 2012).
As the library is publicly accessible on GitHub to the open-source community, the developers are allowed to modify its methods to their own purposes and implement new methods and ports to other DBMS. This is a major advantage over traditional closed-source packages (Hellerstein *et al.* , 2012).

For this purposes some techniques are used by MADlib to succeed in handling the data in the DBMS. The framework consists of the following major components (McQuillan, 2015):

1. C++ database abstraction layer
2. C++ implementation functions
3. Python Driver functions

3.1 C++ Implementation Functions

The functions are the actual implementations of the individual algorithms (ex. Simplex). Included in this C++-implementation are the core functions and the aggregates that are used for the particular algorithms (McQuillan, 2015).

3.1.1 User Defined Aggregates

The first technique used for implementation are user-defined aggregates (UDAs). UDAs are used to implement mathematical functions that take a number of tuples as input variables. A UDA needs to be composed of at least two user-defined functions (UDF) (Hellerstein *et al.* , 2012):

1. **Transition function:** takes the current state and a new tuple and combines them to a new transition state.
2. **Merge function (optional):** takes two current states and merges those transition states by computing a new transition state.
3. **Final function:** takes a old transition state and transforms it into an output value.

3.1.2 State

The state stores all variables used for the implementation of the algorithm. For this purpose a datatype of the state must be defined which can be a vector, a matrix or just one number.

3.2 C++ Database Abstraction Layer for UDFs

This low-level abstraction layer provides matrix operations for easier writing of UDFs. The layer supports three classes of functionality: resource management, type bridging and math library integration. Type briding provides us the c++-data types and functions and additionally translates it to database types and vice versa. The resource management provides a runtime interface to DBMS-managed resources (e.g. allocating/deallocating memory)which separates the c++-exception handling and the DBMS handlers. The final class of functionality is the integration of third party libraries such as Eigen which makes it easier for developers to handle linear-algebra (Hellerstein *et al.* , 2012).

3.3 Python Driver Functions

Most statistical methods implemented in MADlib are iterative which means that a data set needs to be passed many times. For this purpose some variations for implementation came up for MADlib (Hellerstein *et al.* , 2012):

- **Virtual Tables:** A virtual table with n rows can be declared to go through n iterations. Then join this virtual table with a view representing a single iteration. This method can only be used for stateless algorithms, because it isn't possible to pass the state through the iterations.
- **Window Aggregates for Stateful Iteration:** If a current iteration depends on a previous iteration then window aggregates can be used to transport the state through all iterations.
- **Recursive Queries:** Recursive Queries can be used to perform this iteration with user-defined stopping conditions.
- **Python Driver Functions:** Provides control over the iteration of the data movement and creates temp tables to avoid moving data outside of the Database Management System (DBMS) (McQuillan, 2015).

The virtual tables can be used only for predefined number of iterations (they do not support termination conditions) and they cannot be used for stateful algorithms. As the window aggregates and recursive queries don't provide generality for all SQL engines and portability, MADlib has chosen Python Driver Functions to implement complex iterative methods (Hellerstein *et al.* , 2012).

4 Implementation of Simplex Algorithm in PostgreSQL

For the implementation of the revised simplex algorithm, values need to be stored in two different kinds of memories.

4.1 Disk

Matrices of the standard form of the problem are represented as relations in the Linear Program example:

Input relations:

- Coefficients of the constraints: `A (row int, col int, value double precision)`
- The right side of the constraints: `b (row int, value double precision)`
- Cost coefficients: `c (col int, value double precision)`

Generated relations:

- Storing position of relation basis: `basis (position int, i int)`
- Storing position of relation nonbasis: `nonbasis (position int, i int)`

The matrices `A`, `b` and `c` are represented in relations, but the zero values are ommited from the relation to safe storage. For relations `basis` and `nonbasis`, less computations are used if they are stored in the disk instead of storing it in the RAM, because tuples then can be filtered from the relation with a `WHERE`-statement. This means not all values need to be considered for swapping values in relations `basis` and `nonbasis`.

Example For the shown example, the following data is stored in the disk. Relation A is divided into D (white cells) and B (gray cells):

rowi	coli	value
0	0	2
0	1	1
0	2	4
1	0	2
1	1	3
1	2	2
2	0	4
2	1	1
2	2	4
3	0	2
3	1	5
3	2	1
0	3	1
1	4	1
2	5	1
3	6	1

Table 5: Relation A

rowi	value
0	6
1	13
2	12
3	10

Table 6: Relation b

coli	value
0	7
1	9
2	4

Table 7: Relation c

position	i
0	3
1	4
2	5
3	6

Table 8: Relation basis

position	i
0	0
1	1
2	2

Table 9: Relation nonbasis

4.2 State in RAM

For every User Defined Function a previous state is used to calculate the new one. For this we need to pass the state through storage. The state contains the following elements:

Implemented Name	Math Name	Description	Size
numRows	-	Contains the number of rows processed so far	1
nval	n	Describes the number of decision variables, which is the number of columns of matrix D	1
mval	m	Describes the number of constraints, which corresponds to the number of rows of the matrix D	1
bimat	B^{-1}	Corresponds to the B^{-1} matrix	m^2
yvec	y^T	Stores the computed y^T-vector	m
zval	z	Stores the value of the objective function where to find an optimal solution	1
aqvec	$\overline{a}_q$	States the $\overline{a}_q$-vector	m
minRatio	-	Defines the minimum ratio of $\beta/\overline{a}_q$ initialized by -1	1
cbvec	c_B^T	Stores the c_B-vector which contains m-values	m
cdvec	c_D^T	Contains the values of c_D-vector which stores n-values	n
rdvec	r_D^T	Describes the relative cost vector r_D which contains of n-values	n
rval	k	Indicates the value of the pivot row after every iteration	1
qval	q	Indicates the value of the pivot column after every iteration	1
status	status	Describes the status of the algorithm (0=Continue, 1=Unbounded, 2=Optimum)	1

Summed up, the state needs following storage:

$$\begin{aligned}\text{Storage} &= 1+1+1+m^2+m+1+m+1+m+n+n+1+1+1 \\ &= m^2+3m+2n+8 = \underline{\underline{O(m^2+n)}}\end{aligned}$$

The implementation of Simplex always passes the whole state through the aggregate functions. The implementation could be improved using just variables needed by the specific aggregate UDF. Between calling the aggregate UDFs a interstate can be implemented to provide that all attributes can be passed through the aggregate UDFs.

4.3 Implemented UDFs

4.3.1 Compute Relative Costs

Aggregate UDF which computes the relative cost coefficients and determines which variable is having the most negative cost coefficient q.

Transition Function Runs for every tuple of the relation A corresponding to nonbasic variables. In line 1 it first calculates the relative cost vector for the position *coli*. If there is a negative value in r_D which is smaller than the saved *state.qval*, this value is updated.

Listing 1 ComputeRelativeCostsTrans (int rowi, int coli, int value, State state)

```
1: state.rdvec[coli] -= value * state.yvec[rowi];
2: if (state.rdvec[coli] < 0 && state.rdvec[coli] < state.rdvec[state.qval])
3:         state.qval = coli;
4: return state;
```

Final Function Checks if the optimum solution is already found. For this purpose it determines if the vector r_D contains of negative values. If this isn't the case, then it assigns the value 2 to the state, which means the optimal solution is found.

Listing 2 ComputeRelativeCostsFin (int rowi, int coli, int value, State state)

```
1: if (state.numRows == 0)
2:         return Null();
3: if (state.rdvec[state.qval] >= 0)
4:         state.status = 2;
5: return state;
```

4.3.2 Compute Pivot Column

Aggregate UDF which calculates the pivot column $\overline{a}_q$.

Transition Function Computes the pivot column and runs for every tuple of A (nonbasis), `WHERE coli = state.qval`. In line 2 we can see a use of the Eigen library where the algorithm multiplies a row of a matrix by one value.

Listing 3 ComputePivotColumnTrans (int rowi, int value, State state)

```
1: if(state.numRows == 0) state.aqvec.setZero();
2: state.aqvec += state.bimat.col(rowi) * value;
3: return state;
```

Final Function This final function checks if there are only negative values in $\overline{a}_q$ which would mean that the solution is unbounded.

Listing 4 ComputePivotColumnFin (int rowi, int value, State state)

```
1: bool negative = true;
2: for(int i = 0; i < state.mval; i++) {
```

```
3:          if(state.aqvec[i] >= 0) negative = false;
4: }
5: if(negative == true) state.status = 1;
6: return state;
```

4.3.3 Determine Leaving Variable

This UDF calculates the ratios $\beta/\overline{a}_q \; \forall \overline{a}_q > 0$ to determine which vector is leaving the basis.

Transition Function In line 1 the value *state.rval* is initialized to -1. In the for-loop a temp variable stores the current rows ratio and then checks if this value is smaller than the previous ratio. After executing this algorithm, the smallest ratio is stored in *statae.minRatio* and the number of the pivot row is stored in *state.rval*

Listing 5 DetermineLeavingVariableTrans (State state)

```
1: state.rval = -1;
2: for(int i = 0; i < state.mval; i++) {
3:          double temp = state.betavec[i] / state.aqvec[i];
4:          if(state.rval > -1) { //there is a value
5:                  if(temp < state.minRatio) {
6:                          state.minRatio = temp;
7:                          state.rval = i;
8:                  }
9:          } else {
10:                 state.minRatio = temp;
11:                 state.rval = i;
12:         }
13: }
14: return state;
```

4.3.4 Update Tableaux

Transition Function This function updates B^{-1} (with elementary row operations), swaps the correct values of c_b and c_d, and calculates the β-vector. Every line between line 1 and line 18 is executed only once at the beginning of the iteration. In line 2 the pivot row of B^{-1} is updated according to vector $\overline{a}_q$. In line 4 all other rows (except the pivot row) are updated, so that $\overline{a}_q$ is entering the basis. From line 10 - line 12 the swapping of vectors c_b and c_d corresponding to the value *state.rval* and *state.qval* is executed. Then in line 15 vector *state.betavec* is calculated according to the new B^{-1}.

Listing 6 UpdateTableauxTrans (int rowi, int value, State state)

```
1: if(state.numRows == 0) {
2:          state.bimat.row(state.rval) = state.bimat.row(state.rval) /
3:                  state.aqvec[state.rval];
4:          for(int i = 0; i < state.mval; i++) {
5:                  if(i != state.rval) {
6:                          state.bimat.row(i) -= state.aqvec[i] *
7:                                  state.bimat.row(state.rval);
```

```
 8:                 }
 9:         }
10:         double cbtemp = state.cbvec[state.rval];
11:         state.cbvec[state.rval] = state.cdvec[state.qval];
12:         state.cdvec[state.qval] = cbtemp;
13:         state.betavec.setZero();
14: }
15: state.betavec += state.bimat.col(rowi) * value;
16: return state;
```

Final Function This function will be executed only once at the end of the aggregated UDF. As a result we get the new calculated y^T and the updated z. In line 1 the algorithm again takes use of the Eigen library to set all values of the vector equal to zero. In line 2 the new y^T vector is calculated and in line 4 the new z-value gets updated.

Listing 7 UpdateTableauxFin (int rowi, int value, State state)

```
1: state.yvec.setZero();
2: state.yvec = trans(state.cbvec) * state.bimat;
3: state.zval = 0;
4: state.zval = trans(state.cbvec) * state.betavec;
5: return state;
```

4.3.5 Simplex

The aggregated UDFs are executed iteratively using the following algorithm. From line 1 until line 20 the actual algorithm as described in section 2.2 is executed by using the aggregate UDFs explained in section 4.3. In line 21 - line 24 the values are added to a new created table, which stores the results of z and b.

Listing 8 Simplex (Table A, Table b, Table cd)

```
 1: for i in range(0, max_iter):
 2:         state = plpy.execute(SELECT ComputeRelativeCosts (a_row, a_col,
 3:                 a_val, state) FROM A
 4:                 WHERE d_col IN (SELECT i FROM nonbasis))
 5:         if state.status == 2 then break
 6:         state = plpy.execute(SELECT ComputePivotColumn (a_row, a_val)
 7:                 FROM A WHERE a_col =
 8:                 (SELECT i FROM nonbasis WHERE postition = state.qval))
 9:         if state.status == 1 then break
10:         state = plpy.execute(SELECT DetermineLeavingVariable (state))
11:         state = plpy.execute(SELECT UpdateTableaux (b_row, b_val, state))
12:         q_result = plpy.execute(SELECT getqvalue(state))
13:         r_result = plpy.execute(SELECT getrvalue(state))
14:         swap_to_basis = plpy.prepare(SELECT i FROM nonbasis
15:                 WHERE position = q_result)
16:         plpy.execute(UPDATE nonbasis
17:                 SET i = (SELECT i FROM basis WHERE position = r_result)
18:                 WHERE position = q_result)
19:         plpy.execute(UPDATE basis
```

```
20:                 SET i = swap_to_basis WHERE postition = r_result)
21: CREATE TABLE out_z (value DOUBLE PRECISION);
22: INSERT INTO out_z (value) VALUES state.zval;
23: CREATE TABLE out_b (rowi INT, value DOUBLE PRECISION);
24: INSERT INTO out_b (rowi, value) VALUES (rowNumber[], values[]);
```

4.4 Open questions on optimizing the implementation

Would it be more efficient if a tuple of relation A represent a row or columnof matrix A? Actually it sounds logical that it would save storage if an attribute was omitted. In the implementation of Simplex $(n \times m + m)$ rows in the relation are needed.

- $(n \times m)$-rows for the coefficients of the decision variables.
- m-rows for the coefficients of the slack variables.

If this 3 column relation would change to a 2 column relation, storage could be saved. For example the relation could look like `A (row int, value double precision[])` or `A (col int, value double precision[])`. When using this relation design it wouldn't be possible to omit zero values, which would lead to use more storage again. Thus, this idea isn't that good for large sparse matrices. Remembering variable τ which stands for the nonzero values in A, the computation would look like:

- **3 column relation `A (col int, row int, value double precision)`:**

$$2 \text{ int} + 1 \text{ float} = 2 * 4 \text{ Byte} + 1 * 4 \text{ Byte} = 12 \text{ Bytes / tuple}$$

$$\text{For m} \times \text{n tuples} \Rightarrow 12[m \times n] \text{ and because of sparsity } \implies 12 * \tau \text{ Bytes}$$

- **Per row `A (col int, value double precision[])`:**

$$1 \text{ int} + n \text{ float} = 1 * 4 \text{ Byte} + n * 4 \text{ Byte} = 4(n+1) \text{ Bytes / tuple}$$

$$\text{For m tuples} \Rightarrow 4nm + 4m \text{ Bytes}$$

- **Per column `A (row int, value double precision[])`:**

$$1 \text{ int} + m \text{ float} = 1 * 4 \text{ Byte} + m * 4 \text{ Byte} = 4(n+1) \text{ Bytes / tuple}$$

$$\text{For n tuples} \Rightarrow 4nm + 4n \text{ Bytes}$$

In the implementation the per cell relation was implemented. The choice of this database design compared to the 'per column'-design should therefore be more efficient. Given those assumption, following comparison can be calculated:

$$\begin{aligned} 12\tau &< 4nm + 4n \\ 12\tau &< 4n(m+1) \\ \tau &< 1/3n(m+1) \end{aligned}$$

which means that the number of nonzero values needs to be smaller then 1/3 of the matrix, so that the implementation design is more efficient than the per column or the per row implementation.

5 Conclusion

As seen in this paper, MADlib provides a framework to implement in-database analytic methods. The predictive analytics of data in a DBMS is becoming more important wherefore MADlib uses user-defined-aggregates (UDAs) to implement mathematical functions in SQL. Exemplary Simplex was implemented to get an overview about MADlib. Simplex gets a linear optimization problem as an input and outputs optimized costs and the optimal number of inputs per input variable.

In this project I got to know about the MADlib framework which is going to increase its popularity and extended this framework with an auxiliary method. Additionally to the framework, I learned implementing user defined-functions in PostgreSQL and building user-defined aggregates (UDAs). Further on I learned the mathematical handling of the simplex algorithm in its both forms. Unfortunately it was out of scope of the project to test the algorithm on the basis of the Swiss Feed Database, where larger data sets could be used instead of testing it only with self developed examples.

References

Hellerstein, Joseph M., Ré, Christopher, Schoppmann, Florian, Wang, Zhe Daisy, Fratkin, Eugene, Gorajek, Aleksander, Ng, Kee Siong, Welton, Caleb, Feng, Xixung, Li, Kun, & Kumar, Arun. 2012 (Apr.). *The MADlib Analytics Library or MAD Skills, the SQL.*

Luenberger, David G., & Ye, Yinyu. 2016a. Basic Properties of Linear Programs. *Pages 11–31 of: Linear and Nonlinear Programming.* International Series in Operations Research & Management Science, no. 228. Springer International Publishing. DOI: 10.1007/978-3-319-18842-3_2.

Luenberger, David G., & Ye, Yinyu. 2016b. The Simplex Method. *Pages 33–82 of: Linear and Nonlinear Programming.* International Series in Operations Research & Management Science, no. 228. Springer International Publishing. DOI: 10.1007/978-3-319-18842-3_3.

McQuillan, Frank. 2015 (Nov.). *Apache MADlib wiki, MADlib Architecture, https://cwiki.apache.org/confluence/display/MADLIB/Architecture.*