Department of Informatics, University of Zürich

BSc Thesis

Linear Optimization in Relational Databases

Noah Berni

Matrikelnummer: 14-728-166

Email: noah.berni@uzh.ch

December 12, 2017

supervised by Prof. Dr. Michael Böhlen and Georgios Garmpis

University of Zurich[UZH]

Department of Informatics

Acknowledgements

First of all, I want to thank my supervisor Georgios Garmpis for his fantastic support, his guidance and his advices. I also want to thank Prof. Dr. Michael Böhlen for the opportunity to write my bachelor thesis at the Database Technology Group of the University of Zurich and for the interesting topic. Besides, I want to thank my family and friends who are supporting me all the time.

Abstract

The Simplex algorithm finds the values for a number of interrelated variables that optimize a linear objective function while considering a set of linear constraints that must be satisfied. This project implements this algorithm in relational databases by using different approaches. An implementation using functions in PLSQL and different versions of an implementation in C by extending the PostgreSQL kernel are discussed. The focus of the project lies on the versions in C, where tuplestores are used to materialise the intermediate results. A cost formula is deduced for these versions. To compare the efficiency of the implementations, different Linear Programs are solved using the implementations and the consumed times are measured and discussed.

Zusammenfassung

Der Simplex Algorithmus findet die Werte für eine Anzahl zusammenhängender Variablen, so dass eine lineare Zielfunktion unter Einhaltung eines Sets von linearen Nebenbedingungen optimiert wird. Dieses Projekt implementiert diesen Algorithmus in relationalen Datenbanken mit verschiedenen Ansätzen. Eine Implementation, welche Funktionen in PLSQL benutzt, sowie verschiedene Versionen einer Implementation in C, bei welcher der PostgreSQL Kern erweitert wird, werden diskutiert. Der Fokus des Projektes liegt auf den Versionen in C, die „tuplestores“ benutzen, um die Zwischenresultate zu materialisieren. Eine Kostenformel wird für diese Versionen hergeleitet. Um die Effizienz der Implementationen zu vergleichen, werden verschiedene lineare Probleme gelöst und die benötigte Zeit gemessen und diskutiert.

Contents

List of Figures

List of Tables

1. Introduction

The goal of optimization is to find an optimal set of variable values that maximizes or minimizes an objective function while a set of constraints must be satisfied. A linear optimization problem consists of a linear objective function and a set of linear constraints. Such linear optimization problems for example appear in the production of animal feed. The feed mills try to create the perfect food to optimize the efficiency of the livestock. To do so, they try to determine feed compounds that satisfy various restrictions, which are specified by domain experts. A restriction indicates the minimum and maximum containment (g/Kg) of a nutrient in a feed compound. Since different animals have different nutritional needs there are many restrictions for the same nutrient. Animal 'A' needs many proteins and hence the food should fulfil the restriction for high protein food ('high-protein'), while animal 'B' needs less proteins and the food should fulfil the restriction for low protein food ('low-protein'). Different examples for such restrictions are shown in Table 1.1.

Table 1.1.: Nutrient Constraints

diet	nid	min	max
low-lysine	LYS	NULL	4
high-lysine	LYS	4	8
low-protein	CP	NULL	2
medium-protein	CP	2	4
high-protein	CP	4	6
...	...	...	...

With recommendation of domain experts feed mills can select a subset of these restrictions and follow them to create a suitable food compound for a specific animal. To pick the correct mixture of individual foods, data warehouses like the Swiss Feed Database provide information about nutrient content of feeds, which can be used to create an accordant feed compound. An example of the data in the Swiss Feed Database is shown in Table 1.2. The nutrient content of each feed is given by attribute g (g/Kg) of `fact_table`. An additional table with the energy content of the different feeds is shown in Table 1.3.
When we combine Table 1.2 with the restrictions in Table 1.1 we can build accordant linear functions for every nutrient of the feed compound. To do so, we sum up the values of the fact table, which belong to the nutrient we want to create a restriction and compare the resulting term with the restriction we want to satisfy. An example for the restriction of 'low-protein' is shown below. In every kilogram of barley there are three

fid	nid	g
Barley	LYS	1
Barley	CP	3
Hay	LYS	3
Hay	CP	1
Soy	LYS	1
Soy	CP	2
...	...	...

Table 1.2.: Fact Table

fid	DEP
Barley	12
Hay	1
Soy	10
...	...

Table 1.3.: Energy Content in Feeds

grams of crude protein (CP), in every kilogram of hay there is one gram of crude protein and in every kilogram of soy there are two grams of crude protein. In total there need to be less than two grams of crude protein per kilogram, and hence all these three values summed up need to be smaller than two.

$$\text{low-protein: } 3x_{Barley} + 1x_{Hay} + 2x_{Soy} \leq 2$$

It is likely, that different feed compounds satisfy all the chosen restrictions and all of them could be chosen by a feed mill. Normally, nutrient content is critical for good quality food and the corresponding restrictions must be satisfied, while other parameters are just optimal when they are as small or as high as possible. A good example for that is the price of the feed compound. When there are different possibilities to reach a goal, we most likely pick the one, which has the lowest costs for us. Or in the other way, when there are different possibilities with the same cost, we most likely pick the one, which has the biggest output. Therefore the feed mills pick the feed compound, which fulfils all the restrictions and optimizes another value, which increases the efficiency of it. The function to be optimized is the objective function, since we aim to optimize it while holding the restrictions. For example if we want a feed compound with maximized energy content, we can sum up every value in Table 1.3 to build the objective function. The accordant objective function is shown in the following.

$$\text{Maximize DEP} = 12x_{Barley} + 1x_{Hay} + 10x_{Soy}$$

A set of restrictions and an objective function define a full optimization problem. If we want a feed compound with the restriction 'low-protein' for crude protein and 'low-lysine' for lysine, that has maximum energy content, we create the following problem.

$$\begin{gathered}\text{Maximize } z = 12x_{Barley} + 1x_{Hay} + 10x_{Soy} \\ \text{Subject to } 3x_{Barley} + 1x_{Hay} + 2x_{Soy} \leq 2 \\ \text{and } 1x_{Barley} + 3x_{Hay} + 1x_{Soy} \leq 4 \\ \text{where } x_{Barley}, x_{Hay}, x_{Soy} \geqslant 0\end{gathered}$$

We have an objective function along with two restrictions. The amount of barley, hay and soy obviously needs to be at minimum zero, which is defined in the last row. Now that

we have set up our problem we can try to solve it by inserting values in the variables and check if the restrictions are fulfilled. At this position Linear Programs need to be introduced. A Linear Program refers to optimizing a linear objective function under constraints that are expressed as a set of linear equalities or inequalities. It can be defined as in the following equation. The variables `v` and `c` go through all ids of variables (feeds) and constraints respectively. x_v represents the proportion of each feed that comprises the result diet compound.

$$\begin{gathered}\text{Minimize/Maximize: } z = \sum_v (c_v x_v) \\ \text{Subject to } l_c \leq \sum_v (g_{vc} x_v) \leq u_c \\ l_v \leq x_v \leq u_v \\ x_v \in \mathbb{R}, \forall v\end{gathered}$$

Obviously our defined problem is a Linear Program, it has an objective function as well as two constraints each with an upper bound. For Linear Programs, there exists an algorithm called Simplex Method [LY16], which can find the solution for a Linear Program with the optimal objective value. Hence we can apply this method for our problem to find the optimal feed compound.
The main goal of this project is to implement this algorithm in PostgreSQL, that an optimal solution can be found for a Linear Program. To do so, we first need to know how this Simplex algorithm works. In Section 2, the Simplex algorithm is introduced along with solving the initial example step by step for better understanding. Section 3 investigates different schemas, which could be used to store the data of the problem in relational databases to apply the Simplex algorithm and shows their advantages and disadvantages. Section 4 presents a PLSQL implementation of the Simplex algorithm, while in Section 5 the implementation of the algorithm in PostgreSQL is described. Both implementations store the data in the schema chosen in Section 3. In Section 6 the two implementations are compared by the running times and the costs for the PostgreSQL implementation are calculated. At the end a conclusion is given.

2. Introducing Simplex Method

The Simplex algorithm mentioned in the introduction is discussed in this section along with solving the example presented in the introduction. After the example from the introduction is solved, a more complex example is discussed and some additional techniques needed to solve it are presented.

2.1. Preparation for the Simplex algorithm

As mentioned in the introduction, the Simplex algorithm can be applied to solve Linear Programs. The goal of it is to calculate the optimal value of the objective function considering the given constraints. Beside of the optimal value it also delivers the values of the different variables to form this optimal value. The variables represent the amount of feeds, therefore at the end we should know which feed compound we should take to reach the optimal value and what the optimal value is. As a preparation step of the Simplex algorithm the Linear Program needs to be stored in a table, where every constraint as well as the objective function is represented by a row and every variable as well as the bounds by a column. Table 2.1 shows the according table of our initial problem.

Table 2.1.: Initial Problem in Table Format

Constraint ID	Barley	Hay	Soy	RHS
CP	3	1	2	2
LYS	1	3	1	4
optimize	12	1	10	0

By storing the problem in the table format, we have lost some important information. The bounds, which are stored in the column `RHS` are total numbers and we do not know if they are lower or upper bounds or even an exact number(equality) for the nutrient. To prevent this loss of information we need to apply an additional step, before we store the problem in a table. We make equalities out of the inequalities by inserting a slack or a surplus variable. If we have an upper bound and need to add a value, we call it a slack variable. It represents the unused resources of the idle resources and has the coefficient $+1$. If we have a lower bound and need to subtract a value, we call it a surplus variable. It represents the excess amount of resources utilized and has the coefficient -1, because we need to subtract a value and the values of the variables all need to be positive. If a constraint has a lower and an upper bound, two similar equalities need to be

created, which differ only in the right hand side and that one has a slack and the other a surplus variable added accordant to the bound they belong to. Since we have two upper bounds we add a slack variable for both the first and the second row in our example. Beside the type of the bounds we also lost the information if we had a maximization or minimization problem. To avoid this, we consider only minimization problems and if we have a maximization problem, we change the problem to an equal minimization problem. To do so, we multiply the objective function by -1, which changes the type of the problem as well. After applying these changes we get the new problem shown in the following:

$$\begin{gathered}\text{Minimize } z = -12x_{Barley} - 1x_{Hay} - 10x_{Soy} \\ \text{Subject to } 3x_{Barley} + 1x_{Hay} + 2x_{Soy} + 1x_{s1} = 4 \\ \text{and } 3x_{Barley} + 1x_{Hay} + 3x_{Soy} + 1x_{s2} = 3 \\ \text{where } x_{Barley}, x_{Hay}, x_{Soy}, x_{s1}, x_{s2} \geqslant 0\end{gathered}$$

Now that we have only equalities left and that we know that it is a minimization problem we can store the problem in table format without losing any information. The new problem is presented in Table 2.2.

Table 2.2.: Initial Problem With Slack Variables

Constraint ID	Barley	Hay	Soy	s1	s2	RHS
CP	3	1	2	1	0	2
price	1	3	1	0	1	4
optimize	-12	-1	-10	0	0	0

The idea of Simplex is to iteratively decrease the objective value by moving from a feasible solution to another feasible one with a lower objective value. A feasible solution means that every constraint is satisfied by the current values of the variables. A trivial feasible solution is given if there are m columns, that form an identity matrix, where m is the number of constraints. This means that for every constraint row there must be a column with the coefficient value 1 in it, while for the other rows the coefficient value in this column is 0. The variables of these columns are called basic variables. To form the solution each basic variable is assigned to the coefficient value of the `RHS` column of the row its coefficient value is 1. The non-basic variables are assigned to 0. By doing this, every constraint is satisfied and we have a feasible solution. Since the non-basic variables are zero, only the basic variables are building the value of the objective function and hence the solution. In Table 2.2, `s1` and `s2` compose the identity matrix and hence they are the basic variables. `s1` is equal to 2 and `s2` equal to 4. Inserted in the objective function this results in the value 0, since the variables `s1` and `s2` do not change the objective value. The additive inverse of this value is stored in the cell of the `optimize` row and the `RHS` column. In problems with only smaller than conditions, there is always a basic feasible solution, since the real variables can all be 0 and the slack variables can be used as basic variables, because they always have the coefficient $+1$ and are contained in only one constraint. In these cases, the initial feasible solution

has the objective value 0. However, when equal or bigger than conditions appear in the constraints no slack variable is added and possibly no basic feasible solution is available. Techniques to find an initial feasible solution when no trivial solution is given at the beginning are explained later with a more complex example. For our initial problem there is a basic feasible solution and we can try to find the next feasible solution with a lower objective value by executing a so called Simplex step.

2.2. Simplex Step

Now that we have an initial feasible solution with the slack variables as basic variables we can start with the Simplex algorithm. Like mentioned the Simplex algorithm tries to decrease the objective value by moving from a feasible solution to another feasible solution with a lower objective value. To do so the Simplex algorithm exchanges a basic variable with a non-basic variable, which means a basic variable is set to 0, while a non-basic variable is increased and hence it becomes the new basic variable. This step to exchange a basic variable and to decrease the objective value is called a Simplex step. To find out which non-basic variable should be increased, so that the objective value is decreased the most, we look at the `optimize` row, which stores the objective function. The objective function contains the relative cost coefficients. This means that in this row we can see how the objective value changes when we increase a non-basic variable from 0 to a positive value, while an other basic variable will be set to 0. This can be explained very well by looking at Table 2.2. We have a feasible solution with the initial basic variables `s1` and `s2`, consequently the objective value is 0. Considering the objective function of our example, the value of `z` decreases by 12 if x_{Barley} is increased by 1, by 1 if x_{Hay} is increased by 1 and by 10 if x_{Soy} is increased by 1. Now the first step in the Simplex algorithm is to pick the one column with the highest negative value, since `z` can be decreased the most when the variable of this column is increased. We call this column pivot column. If there are two columns, which can be the pivot column, it can be chosen one of these randomly. We mainly pick the one, which was considered first. If there is no negative value, `z` can not be decreased any more and the target of the Simplex algorithm (optimal value of `z`) is reached. In our example `Barley` is the pivot column, since -12 is the most negative coefficient value.
Now that we know which variable we want to increase in this step, we need to know how much we can increase it that all constraints still hold. We divide the coefficient value in the right hand side (`RHS` column) through the coefficient value in the pivot column and get the ratio of them for each constraint row. Such a ratio represents the value, which the variable in the pivot column can be increased without breaking the corresponding constraint. If the ratio in a row is negative the pivot variable can be increased infinitely regarding this constraint. Therefore we pick the smallest non-negative ratio, because the pivot variable can be increased by that value without breaking any constraint. The row, to which the smallest positive value belongs, is called the pivot row. If there are only negative ratios, the pivot variable can be increased infinitely regarding each constraint, which means that the problem is unbounded. Since the coefficients in the right hand

side are assigned to the basic variables, which are all non-negative, they need to be non-negative as well. Therefore we can only consider the value in the pivot column to find out if the ratio is negative. If the value in the pivot column is negative or zero, this row can not be considered as the pivot row.
In our example there are only two constraint rows, `CP` has a ratio of 2/3 and `LYS` a ratio of 4. 2/3 is the smallest one and hence `CP` is the pivot row. Similar to the pivot column, it can be randomly chosen, if two rows have the same ratio. If a pivot row is found, the element at the position of the pivot column in this row is called the pivot element.
Now we need to transform the initial table into an equivalent one that the coefficient of the pivot element is 1 and the other coefficients of the pivot column are 0. So that the pivot column becomes part of the basis identity matrix. To manage that, the pivot row is first divided through the pivot element, that the new coefficient value of the pivot element is 1. Since the other rows should have the coefficient value 0 in the pivot column afterwards, we can apply Gaussian elimination. The new pivot row is subtracted as many times from the other rows, that the coefficient value in the pivot column is 0 afterwards. Through that, the basic variable at the pivot row gets replaced by the variable of the pivot column and is set to 0.
With all these steps a zero variable, the pivot column, is increased and turned to a basic variable and a positive variable is set to zero. The Simplex table is transformed so that it represents an equivalent linear problem. But after the transformation the identity matrix corresponds to the new solution, which is improved. This step is repeated until no pivot column can be found, which means the optimum is reached, or until no pivot row can be found, which means the objective function is unbounded and no optimal solution exists. In both cases the Simplex step is finished and the according result can be found in the last table.
In our example we already have found out that `Barley` is the pivot column and `CP` is the pivot row, hence the coefficient value of the pivot element is 3. The values in the pivot row are divided through 3. The other rows need to subtract the pivot row as many times that their coefficient value in the pivot column is 0. The row `LYS` needs to subtract the pivot row once and the row `optimize` 12 times. Table 2.3 shows the new table after it is updated like described. We can see that `Barley` is now a basic variable with the value 2/3 and the objective value decreased from 0 to -8. (additive inverse of 8) The reduction of the objective value is exactly 2/3 from -12, which was the relative cost coefficient of `Barley` in Table 2.2.

Table 2.3.: Updated Table After one Simplex Step

Constraint ID	Barley	Hay	Soy	s1	s2	RHS
CP	1	1/3	2/3	1/3	0	2/3
LYS	0	8/3	1/3	-1/3	1	10/3
optimize	0	3	-2	4	0	8

Now that we finished one Simplex step, we can go back to the start of the loop and try to decrease the objective value again by applying another Simplex step. We start

again by looking for a pivot column. In the relative cost coefficients row (`optimize`) is only one negative coefficient value left, which is in the column `Soy`. Hence, increasing the value of `Soy` is the only way to decrease the objective value and it is the new pivot column. The ratios are 1 for `CP` and 10 for `LYS`. Therefore `CP` is the new pivot row and we should be able to increase `Soy` by 1 according to this constraint and hence decrease the objective value by 2. Table 2.4 shows the new table after updating it the same way as before. We can see that `Soy` is 1 and the new objective value is indeed -10 (additive inverse of 10), which confirms our calculations.

Table 2.4.: Updated Table After two Simplex Steps

Constraint ID	Barley	Hay	Soy	s1	s2	RHS
CP	3/2	1/2	1	1/2	0	1
price	-1/2	5/2	0	-1/2	1	3
optimize	-3	-4	0	-5	0	10

Again we finished one Simplex step and we can start at the beginning of the loop. There is no more negative value in the relative cost coefficients row and therefore we can not decrease the objective value any more and we have reached the optimum. By assigning the coefficient value at the right hand side of each row to the corresponding basic variable we can find out the values of the different variables as we did before to find the feasible solutions. If a variable is a non-basic variable its value is 0. The additive inverse of the optimal value is still presented in the objective function row at the right hand side. Hence the optimal value of this problem is -10. Since we multiplied the objective function by -1 at the beginning to create a minimization problem, we need to multiply the optimal value by -1 again to get the correct value. Hence the optimal value is 10 and it can be reached by setting the values Barley = 0, Hay = 1 and Soy = 0.

2.3. Introducing two-phase method

As already told there exists some more complicated examples, which can not be solved that easily, since there is no basic feasible solution right at the beginning. When a lower bound or an equal condition appears, there is no slack variable, which can serve as basic variable and build the identity matrix. However, to apply the simplex method we need a feasible solution to start, hence we use a new technique, the so called two-phase method. Like the name says, this method has two phases. In the first phase the method searches for a feasible solution and in the second phase, if a feasible solution is found, it is optimized using the same technique as used for the first problem with a basic feasible solution. Regarding a new example shown in the following:

$$\begin{gathered}\text{Minimize } z = 4x_{Barley} + 1x_{Hay} + 1x_{Soy} \\ \text{Subject to } 2x_{Barley} + 1x_{Hay} + 2x_{Soy} = 4 \\ 3x_{Barley} + 3x_{Hay} + 1x_{Soy} \geqslant 3 \\ \text{where } x_{Barley}, x_{Hay}, x_{Soy} \geqslant 0\end{gathered}$$

Again as a first step we need to make equalities out of the inequalities. We need to add a surplus variable with the coefficient -1 to the second constraint row. Since we already have a minimization problem we can let the objective function as it is. This leads to the new problem, which is shown in table format in Table 2.5. We can see that neither for the first and for the second row there is a basic variable and hence we do not have a basic feasible solution.

Table 2.5.: Table Format of Second Problem With Surplus Variable

Constraint ID	Barley	Hay	Soy	s1	RHS
CP	2	1	2	0	4
LYS	3	3	1	-1	3
optimize	4	1	1	0	0

To find a feasible solution, in the first phase of the two-phase method an artificial problem is created, whose optimal solution, if there exists one, is a feasible solution of the original problem. For every row, where no basic variable is available, an artificial variable with the coefficient $+1$ is added, which can serve as identity matrix column and hence as initial basic variable. These artificial variables are fictitious and have no physical meaning. For example in the first constraint we create the following equality out of the initial equality.

$$2x_{Barley} + 1x_{Hay} + 2x_{Soy} + 1x_{a1} = 4$$

Since this term should be equal without considering the artificial variable `a1`, `a1` needs to be zero at the end. The same counts for `a2`. To check if the artificial variables are zero at the end, we build an artificial objective function with the artificial variables inside it and put the initial objective function at the side for the moment. If the optimal value of this artificial objective function is zero, the artificial variables are zero as well. The artificial problem is presented in the following:

$$\begin{gathered}\text{Minimize } z = x_{a1} + x_{a2} \\ \text{Subject to } 2x_{Barley} + 1x_{Hay} + 2x_{Soy} + 1x_{a1} = 4 \\ 3x_{Barley} + 3x_{Hay} + 1x_{Soy} - 1x_{s1} + 1x_{a2} = 3 \\ \text{where } x_{Barley}, x_{Hay}, x_{Soy}, x_{a1}, x_{a2} \geqslant 0\end{gathered}$$

This new artificial problem has a basic feasible solution with the artificial variables as basic variables. To find a feasible solution for the original problem, `a1` and `a2` need to be zero. To do so, we need to minimize the artificial problem. If the optimal value of the artificial problem is zero, the final table of the artificial problem represents a feasible solution for our initial problem, if we remove the columns of the artificial variables. So, after removing the artificial variables we run the second phase of the algorithm, which searches for the optimal value of the original problem.
To find the minimum solution of the artificial problem we can apply the already known Simplex algorithm. Before we start running the Simplex algorithm we first need to compute the relative cost coefficients. Since in the Simplex algorithm, the objective function row should stand for the relative cost coefficients, the coefficient value of each

basic variable needs to be zero in this row, because these variables can not be increased in the next Simplex step. Because of this, we need to transform the artificial objective function row accordingly. To do this, we need to subtract the other rows from the artificial objective function row as many times that the coefficient values of the basic variables are zero in the artificial objective function. Since each row belongs to one basic variable, we principally need to subtract every row as many times as the coefficient value of the corresponding basic variable is in the artificial objective function. For example the first row needs to be subtracted once from the artificial objective function, because the coefficient value in the column `a1`, which is the basic variable for this row, is one in the artificial objective function. In the first phase these values are always one for the artificial variables, because the artificial objective function pretended that.
The updated relative cost coefficients row for the artificial problem is shown in Table 2.6. In the `artObjective` row in the `RHS` column also the additive inverse of the artificial objective value is presented. The current artificial object value is seven, which is correct since $\texttt{a1} = 4$ and $\texttt{a2} = 3$ and $\texttt{a1} + \texttt{a2} = 7$.

Table 2.6.: Transformed Artificial Objective Function to the Relative Cost Coefficients

Constraint ID	Barley	Hay	Soy	s1	a1	a2	RHS
CP	2	1	2	0	1	0	4
LYS	3	3	1	1	0	1	3
optimize	4	1	1	0	0	0	0
artObjective	-5	-4	-3	1	0	0	-7

Now that the relative cost coefficients of the basic columns are 0, the objective function shows the correct relative cost coefficients and the artificial problem can be optimized using the Simplex algorithm, which is the same algorithm used to optimize the first example, where a basic feasible solution was directly available through slack variables. First we search the pivot column, then the pivot row and finally update the table accordingly that the pivot variable will be part of the basic variables. One additional rule needs to be applied when searching for the pivot row when there are more than one row with the same ratio. If one of these rows has an artificial variable as current basic variable, then this row is preferred before the others without an artificial variable as basic variable, because we want to remove the artificial variables from the basic variables. In our example we can execute two Simplex steps until there is no negative coefficient value in the relative cost coefficients row any more and hence no pivot column. The 2 intermediate tables each after one Simplex step are shown in Table 2.7. In the first step the pivot element is (LYS/Barley) in Table 2.6 and in the first table in Table 2.7 the pivot element is (CP/Soy).
Now the artificial problem is optimized and we can check if the optimal value of the artificial problem is zero. If it is not zero, the initial problem has no feasible solution, which fulfils all the restrictions. In this case the algorithm is finished. As we can see in the second table in Table 2.7, the optimized value of the artificial objective function is zero and we have found a feasible solution for the initial problem. At this position it is

Table 2.7.: Intermediate Optimize Steps

Constraint ID	Barley	Hay	Soy	s1	a1	a2	RHS
CP	0	-1	4/3	2/3	1	-2/3	2
LYS	1	1	1/3	-1/3	0	1/3	1
optimize	4	1	1	0	0	0	0
artObjective	0	1	-4/3	-2/3	0	5/3	-2

Constraint ID	Barley	Hay	Soy	s1	a1	a2	RHS
CP	0	-3/4	1	1/2	3/4	-1/2	3/2
LYS	1	5/4	0	-1/2	-1/4	1/2	1/2
optimize	4	1	1	0	0	0	0
artObjective	0	0	0	0	1	1	0

possible that the basic variables still contain artificial variables with the coefficient value zero at the right hand side. In such a case we can just replace the artificial variable as basic variable with another non-artificial variable, since the value of this basic variable is any way zero and hence the solution does not change. To do so, we can execute another Simplex step, but instead of first looking for a pivot column, the row, where the artificial variable is the basic variable, is chosen as the pivot row and a corresponding pivot column is searched afterwards. In the pivot row we can pick any non-artificial variable with a positive coefficient value in the pivot row as pivot column and if no variable with a positive coefficient value is available also variables with negative coefficient values in the pivot row can be picked. When we have a pivot column and a pivot row the updating of the table is done as in a normal Simplex step. Like this the pivot column will replace the artificial variable as the basic variable. By doing this both variables stay zero. If only zero coefficient values are contained in the non-artificial variables of the pivot row and hence no variable to serve as pivot column can be found, the row does not contain any information for the real problem any more, since the artificial variables do not affect the real objective function and the row can be deleted out.
Back to our problem, now that we have found a feasible solution, the first phase is finished and we can start with the second phase of the algorithm and search the optimum of the real problem. Since the artificial objective function and the artificial variables have no real meaning, they do not need to be considered any more and we take back our original objective function. Table 2.8 shows the original problem with the detected feasible solution.

Table 2.8.: Feasible Solution for the Original Problem

Constraint ID	Barley	Hay	Soy	RHS
CP	0	-3/4	1	3/2
LYS	1	5/4	0	1/2
artObjective	4	1	1	0

At this state we have our initial problem with the updated values and with the current basic variables `Barley` and `Soy`. Since we have new basic variables, we need to transform the objective function that it stores the relative cost coefficients and that the coefficients of the basic variables are zero, similar to the first phase. We again subtract the according rows as many times from the objective function that the coefficients of the basic variables are zero. The new table with the updated cost coefficients is shown in Table 2.9. Now it has a feasible solution and it does not contain artificial variables any more.

Table 2.9.: Feasible Solution With Correct Cost Coefficients

Constraint ID	Barley	Hay	Soy	RHS
CP	0	-3/4	1	3/2
LYS	1	5/4	0	1/2
optimize	0	-13/4	0	-7/2

At this point that we have a feasible solution, we can apply the Simplex algorithm to optimize the real problem. At the current stage the objective value is 3.5 (additive inverse of -3.5), but there are still negative coefficients in the relative cost coefficients row and we can run another Simplex step. The pivot element is (LYS/Hay) and thereby the table gets updated by considering `Hay` as new basic variable. The result of this Simplex step in Table 2.10 is also the final result and we have an optimal solution, since there is no negative value left in the relative cost coefficients row. Again here exists the possibility that the solution is unbounded, when no pivot row can be found.

Table 2.10.: Final Result of the Problem

Constraint ID	Barley	Hay	Soy	s1	RHS
CP	3/5	0	1	1/5	9/5
LYS	4/5	1	0	-2/5	2/5
optimize	13/5	0	0	1/5	-11/5

The minimum of 11/5 can be reached by setting the values Barley = 0, Hay = 2/5 and Soy = 9/5. At this point we know how to apply the Simplex method and how to optimize Linear Programs. Now we want to implement exactly this algorithm using PLSQL and PostgreSQL and compare the different implementations.

3. Different Schemas during the algorithm

In the previous section we have learnt how the Simplex algorithm works and how the problem table is accessed and modified during the algorithm. Before we can start with any implementation we need to decide how we will store the problem table in a relation, that we can work with it as efficient as possible. There are many possibilities how the table could be stored. In this section, different schemas are reviewed and advantages and disadvantages of them are discussed. The table, which we have chosen to use for the implementations, is shown at the end of this section.
As a first possibility we can have a relation, where one row of the table is stored in one tuple of the relation. The coefficient values of the variables are stored in an array in one column of the relation, while the coefficient value at the right hand side and the name of the constraint are stored in a separate column. In Table 3.1 the problem from the introduction is presented using this approach. This schema is quite intuitive and easy to understand, which can be mentioned as a positive point. Regarding the implementations in relational databases, in this schema it is very easy to find the pivot column, because we need to only consider the tuple, which stores the objective function row and look for the most negative value inside the array with these values. However, to find the pivot row always a tuple with the whole row need to be loaded into the memory, which can grow by size when there are many variables, although we need only two values of each row. In this way, values which are not needed at the moment are read and loaded into the memory. This leads to the point that comparing values from different rows is inefficient, because always the whole row need to be loaded. Another point, which should be mentioned, is that an additional table need to be added, where the information of the column names is stored. This is neither a real advantage nor a disadvantage.

Table 3.1.: Tuple per Row Schema

Constraint ID	Values	RHS
CP	3, 1, 2, 1, 0	2
price	1, 3, 1 , 0, 1	4
optimize	-12, -1, -10, 0, 0	0

Instead of storing every row in a tuple it is also possible to store every column in a tuple. In Table 3.2, the problem table from the introduction is presented again, but this time the tuple per column schema is used. Using this schema has similar advantages and disadvantages like the tuple per row schema. Instead of the pivot column it is easy to

find the pivot row in this schema, since only the RHS-tuple and the pivot column-tuple need to be considered. Another similarity to the per row schema is that we need to load unnecessary information into the memory to compare two values. In this schema this problem appears when two values of different columns are compared, for example to find the pivot column. In this case all values of the columns need to be loaded in memory, although we only need the values from the objective function. This is a huge disadvantage for these schemas. Another table with the information of the row names need to be added as done in the tuple per row schema for the column names. Since the values inside the arrays are sorted, a special advantage for this schema appears regarding the two-phase method. The first and the second phase have the main difference, that the first phase considers the `artObjective` row and the second phase the `optimize` row as objective function. In this schema it is possible to put the current objective function at the end of the array and always looking for the last element, which describes the current objective function. Like this it is very easy to access the correct objective function and to differ between the two phases. Since a column is stored in one tuple, it is also very easy to add artificial variables by just adding a new tuple.

Table 3.2.: Tuple per Column Schema

Variable	Values
Barley	3, 1, -12
Hay	1, 3, -1
Soy	2, 1, -10
s1	1, 0, 0
s2	0, 1, 0
RHS	2, 4, 0

In the tuple per row schema it is easy to get the pivot column, while in the tuple per column schema it is easy to get the pivot row. In a hybrid schema these advantages can be combined. A possibility for such a hybrid schema is shown in Table 3.3. This schema is mainly built like the tuple per column schema, but the objective function is separated from the columns. Through that the pivot column can be found by reading the tuple, which stores only the objective function and still the pivot row can be found by only considering the tuples, which store the pivot column and the RHS column. However, to compare two values of two different columns the problem stays that we need to load the whole columns into the memory. Additionally, this schema is probably not that intuitive at the first view, and it can be confusing some times. The information of the rows and the columns both are stored in 2 separate tables.
At this point we want to introduce the schema we have chosen for the implementations. Instead of storing a whole row or column in a tuple we can just store one value of the table in one tuple to avoid the problem of loading unnecessary data into the memory. A part of the problem table from the introduction is shown in Table 3.4 using this approach. To individualize and identify a value, we store it along with the restriction (row) and the variable (column) it belongs to. The main advantage of this schema is that the schema

Table 3.3.: Hybrid Schema

ID	Values
Barley	3, 1
Hay	1, 3
Soy	2, 1
s1	1, 0
s2	0, 1
RHS	2, 4
optimize	-12, -1, -10, 0, 0, 0

is really flexible, because every value can be accessed individually in a tuple. Thereby it is very easy to extend the table by new values or just delete some of them. If one value needs to be read or changed, the tuple with the corresponding value can be handled individually. However, there can exists a huge amount of tuples, because every value is stored individually. This can make it expensive to read a single row, because every tuple needs to be found in the relation and checked individually. This can be prohibited with an index scan by jumping directly to the necessary tuples. The ordering of the tuples can help to do that, more on that will be discussed in Section 5. Summarized, in this schema values can be added or updated easily, but size may be increased.

Table 3.4.: Per Value Schema

row	col	val
CP	Barley	3
CP	Hay	1
CP	Soy	2
CP	s1	1
CP	s2	0
CP	RHS	2
...	...	...

As a final conclusion it is hard to say, which schema is the best one, since all of them have their advantage and disadvantages. We have chosen the tuple per value schema, because it is the most flexible one and we never need to store a whole row or column inside the memory. Beside this, the main reason we picked the per value schema is because the input data in the Swiss feed database (Table 1.2) is stored in the same way and by using the same schema we can directly use this data without changing the schema of it, which takes a lot of time and can be an error-prone process.

4. Implement Simplex using PLSQL

Now that we have decided which schema we will use to store the problem table as relation we can start with the implementations. The Simplex algorithm along with the two-phase method introduced in Section 2 is implemented in this Section using PLSQL. Again we start with implementing the Simplex algorithm itself and present the additional steps for the two-phase method afterwards. At the end, the user should be able to run the query in Listing 4.1 and get the result of the linear optimization problem as a table at the end.

Listing 4.1: Final Query

```
SELECT optimizeLinearProblem('problemTable');
```

4.1. Input Table

As presented in Listing 4.1 the PLSQL function `optimizeLinearProblem` takes as parameter a name of a table. This table is given to the function by the user and on this table the Simplex algorithm will be executed. This input table should have similar content like the problems in the introduction of the Simplex algorithm with different restrictions and one objective function. In Section 3 we have chosen to use the per value schema, which means that for every value in the table one tuple is generated. The input table needs to be given in this schema. Slack and surplus variables need to be already contained in the input table that we do not need to consider whether a constraint has an upper or a lower bound. Since we can not fully control the ordering of the tuples in PLSQL, because the table is shuffled after every update statement, it can not be expected that the ordering stays the same while working with the table, unless we explicitly specify it in every query we execute. Therefore the initial ordering of the values can be randomly. However, the tuples of the objective function need to be stored either with the keyword 'optimize' or with the number '0' in the 'row'-column of the table, that we know which tuples belong to the objective function. Similarly the tuples of the right hand side need to be stored either with the keyword 'RHS' or with the number '0' in the 'col'-column, so that we can differ these tuples from the normal variables.
Since we need to know the basic variable of every row during the Simplex algorithm that we know which variable is assigned to which coefficient value of the `RHS` column, we create a new table at the beginning and store the initial basic variables inside it. In this table we store the name of the rows along with the corresponding basic variable. At the beginning the basic variables are always the slack variables if only lower than conditions

are existent, hence we can store the rows with the corresponding slack variable in this new table. To do so we insert all columns into the basic variables table, where only one value of the column is not equal to 0 and this non-zero value is exactly 1. These columns represent slack variables. Along with a column we insert the row, in which the non-zero value appears and hence to which the slack variable belongs. Listing 4.2 shows the corresponding code.

Listing 4.2: Getting the initial basic variables

```
INSERT INTO basicVariables
   SELECT row, col FROM problem_table
   WHERE val = 1
   AND (SELECT COUNT(*) FROM problem_table AS a1
      WHERE a1.val <> 0 AND a1.col = problem_table.col
      AND col <> 'RHS') = 1;
```

To the cases, where no slack variables are present, we will come back later. The composition of the table with the basic variables is shown in Table 4.1. The basic variable in the row `CP` is `s1` and the basic variable in the row `LYS` is `s2`.

Table 4.1.: Basic Variables Table

row	col
CP	s1
LYS	s2

In extreme cases it is possible that more than one column could be the slack variable for the same row. In such a case one of these columns is redundant and can be deleted out, since it has any way no influence on the solution.

4.2. Simplex Step in PLSQL

Now that we have discussed the input table more deeply and detected the initial basic variables, we can start to implement the Simplex algorithm and search for the optimal value of the objective function regarding the constraints. At first the code of the Simplex algorithm is presented in Listing 4.3, each function used there will be discussed afterwards. This function is called by the main function `optimizeLinearProblem`, where everything is initialized and finally the correct values are returned. Since the main function does just initialize the basic variables, call the Simplex algorithm and return the results, it is not presented here for simple problems. However, for more complex problems, where the two-phase method needs to be applied, it does some other tasks and hence it is shown in Listing 4.8 in the part about implementing the two-phase method in Section 4.3. Also this code from the Simplex algorithm is only a first version of the final code and it will be improved later with some parts according to the two-phase method. However, the concept of this function will stay exactly the same, just some

small changes need to be added later. As a first thing we want to discuss the return value of the function. It is of the type `SimplexState`. This type is an enumerated type and can have the value `optimal`, `unbounded` and `not_feasible`. It stores the type of the result of the linear problem, if the problem has an optimal solution, if the solution is unbounded, or if the problem has no solution. With this information we can later on return the correct values in the main function. The third possibility can not appear for simple problems with only smaller than conditions, since in this case there is always a possible solution.

Listing 4.3: Optimize Function

```
CREATE OR REPLACE FUNCTION optimize()
RETURNS SimplexState AS
$$
DECLARE
   pc VARCHAR;
   pr VARCHAR;
BEGIN
   WHILE TRUE LOOP
      -- getting the pivot column or null if no value
      -- is negative in the objective function
      pc = getPivotColumn();
      -- if no negative values,
      -- the optimum value is found
      IF pc IS NULL THEN
         RETURN 'optimal';
      END IF;
      -- getting the pivot row or null
      -- if there is no non-negative ratio
      pr = getPivotRow(pc);
      -- if no non-negative ratio,
      -- the solution is unbounded
      IF pr IS NULL THEN
         RETURN 'unbounded';
      END IF;
      -- update the rows according to the pivot element
      Perform updateRows(pc, pr);
      -- add the pivot column to the basic variables
      -- with the pivot row as row
      Perform updateBasicVariables(pc, pr);
   END LOOP;
END;
$$ LANGUAGE PLPGSQL;
```

The first step in the Simplex algorithm is to find the pivot column. In the PLSQL implementation we call the function, which does this `getPivotColumn`, it is shown in Listing 4.5. It declares a cursor over every negative value in the objective function row,

which represents the relative cost coefficients. It only takes the negative values, because if the value is positive, `z` is increased when the corresponding column variable is increased and a current basic variable is set to 0. To check if the value is smaller than 0, we use the function `isSmallerThan`, instead of using a simple lower-than-operator. We use this function, because computers can make small calculation errors and hence we want to set a tolerance, in which area two values are handled as the same. Therefore, the function call `isSmallerThan(val, 0)` returns `true`, if `val` is more than the tolerance smaller than 0. Similar functions are used to check for equality and if a value is bigger than another. The value of the tolerance needs to be set by the user before the main function is called by executing the code in Listing 4.4.

Listing 4.4: Setting the tollerance

```
SET simplex_conf.tollerance TO 1e-6;
```

Over the resulting values of the cursor the function is applying a general find minimum algorithm, where always the smallest number until now is stored and then compared with the new values. To compare two values, we again use the function `isSmallerThan`. Finally the stored value is returned, because it is the smallest value of all. If there is no negative value, `z` can not be decreased any more and the target of the Simplex algorithm is reached. The function `optimize` in Listing 4.3 returns the value `optimal` as `SimplexState`.

Listing 4.5: Getting the pivot column

```
CREATE OR REPLACE FUNCTION getpivotcolumn()
  RETURNS character varying AS
$$
DECLARE
   pc VARCHAR;
-- cursor over the negative values in the objective function
   cur CURSOR FOR
      (SELECT col, val FROM problem_table
      WHERE row = 'optimize'
      AND isSmallerThan(val, 0)
      AND col <> 'RHS');
   r RECORD;
   pcRecord RECORD;
   firstRecord BOOLEAN;
BEGIN
   firstValue = TRUE;
   -- find the minimum value stored in the cursor
   FOR r IN cur LOOP
      IF firstRecord THEN
         pcRecord = r;
         firstRecord = FALSE;
      ELSE
```

```
        IF isSmallerThan(r.val, pcRecord.val) THEN
           pcRecord = r;
        END IF;
     END IF;
   END LOOP;

   IF firstRecord THEN
        -- no negative value found
        RETURN NULL;
      ELSE
        pc = pcRecord.col;
        RETURN pc;
   END IF;
END;
$$ LANGUAGE PLPGSQL;
```

Now that we have found the pivot column, we need to find the according pivot row. To find it in our PLSQL implementation, we first calculate the ratios of the right hand side and the pivot column. Listing 4.6 shows the corresponding `SELECT` statement to get these ratios. The first inner query selects all tuples in the `problem_table`, which are belonging to the right hand side (`RHS`) and of which the row is a constraint row (all but the objective function row). This query is joined with a similar table by the name of the row. Instead of the right hand side column the other query selects the tuples, which are belonging to the pivot column. The values in the pivot column need to be positive. If all values in the pivot column are negative, no pivot row can be found and the problem is unbounded. In this case the function `optimize` returns the value `unbounded` as `SimplexState`. Otherwise, the main `SELECT` statement returns the two values from the inner queries divided through each other, which results in the ratios, along with the name of the according row. This whole query can be put into a cursor, which goes through the returned tuples and pick the one with the smallest ratio. The according row is the pivot row.

Listing 4.6: Calculating the Ratios

```
SELECT a1.row, a1.val / a2.val AS ratio
FROM (
   -- all tuples from the right hand side
   SELECT *
   FROM problem_table
   WHERE col = 'RHS'
   AND row <> 'optimize'
   ) AS a1
INNER JOIN (
   -- all tuples from the pivot column with a positive value
   SELECT *
   FROM problem_table
```

```
   WHERE col = pc
   AND isBiggerThan(val, 0)
   AND row <> 'optimize'
   ) AS a2
ON a1.row = a2.row;
```

At this point we have our pivot element and we can start to update the rows. Both the updating of the pivot row and the updating of the other rows is done in the function `updateRows`. Simple `UPDATE` statements can accomplish the task, they are presented in Listing 4.7. To update the pivot row, we just divide the current value through the pivot element. To update the other rows, we store the new pivot row and the pivot column in a separate table, because we need to access them many times and like this we have lower costs to access them. By using a cursor over every row but the pivot row, we update the rows one by one individually. For every row we first get the value in the pivot column of the current row and store it in the variable `valueInPivotColumn`. If this value is not zero we update the tuples of the row by subtracting the product of the variable `valueInPivotColumn` and the value in the pivot row of the column of the current tuple. We only need to update the rows, where the variable `valueInPivotColumn` is not equal to zero and in these rows only the tuples, where the according value in the pivot row is not equal to zero as well, because otherwise the value does not change. By not updating the values, which does not change, we can save a lot of time.

Listing 4.7: Update rows

```
-- get pivot Element
SELECT val INTO pivotElement
FROM problem_table
WHERE row = pr AND col = pc;

-- update pivot row
UPDATE problem_table
SET val = val / pivotElement
WHERE row = pr AND val <> 0;

-- create table with the updated pivot row
CREATE TABLE pivot_row(col VARCHAR, val FLOAT);
INSERT INTO pivot_row
   SELECT a1.col, a1.val
   FROM problem_table AS a1
   WHERE a1.row = pr;

-- create table with the pivot col
CREATE TABLE pivot_col(row VARCHAR, val FLOAT);
INSERT INTO pivot_col
   SELECT a1.row, a1.val
   FROM problem_table AS a1
```

```
   WHERE a1.col = pc;

-- update the other rows (all but the pivot row)
OPEN allRows FOR
   SELECT DISTINCT a1.row FROM problem_table AS a1
   WHERE a1.row <> pr;
FETCH allRows INTO r;
WHILE r IS NOT NULL LOOP
   SELECT val FROM pivot_col
   WHERE row = r.row INTO valueInPivotColumn;
   IF valueInPivotColumn <> 0 THEN
      UPDATE problem_table SET val = val -
      (SELECT a1.val FROM pivot_row AS a1
         WHERE a1.col = problem_table.col) * valueInPivotColumn
      WHERE problem_table.row = r.row
      AND NOT isEqualTo((SELECT a1.val FROM pivot_row AS a1
         WHERE a1.col = problem_table.col), 0);
   END IF;
   FETCH allRows INTO r;
END LOOP;
CLOSE allRows;

DROP TABLE pivot_row;
DROP TABLE pivot_col;
```

At the end of a Simplex step also the table of the basic variables is updated. The basic variable at the pivot row gets replaced by the variable of the pivot column. In the function `optimize` all these functions are inside a while loop, which is running until no pivot column or no pivot row can be found, which means the current problem is optimal or unbounded.
In our initial example, the loop is executed twice until no pivot column could be found any more and `optimal` can be returned as `SimplexState`. In the main function we than know that we have an optimal solution. We can return the optimal value along with the values of the different columns. The returned table is shown in Table 4.2. Instead of creating a really complex query, this return table can be created with the help of the basic variables. The basic variables table, which is created at the beginning and updated during every Simplex step stored every basic variable with the accordant row. Hence we can just read the right hand side of the table and assign the read value to the variable, which belongs to this value accordingly to the basic variables table. The resulting table can be returned.
At the end, the used tables are dropped and the result is shown.

Table 4.2.: Returned Table in PLSQL Implementation

col	val
hay	1
barley	0
hay	0
optimum_value	10

4.3. Two-phase method in PLSQL

As already seen in Section 2 there exists some more complicated examples, which can not be solved that easily, since there is no basic feasible solution right at the beginning. At this point it seems meaningful to look at the main function `optimizeLinearProblem` in Listing 4.8. The function `getBasicVariables`, which is called at the beginning, was already used in the simple example to detect the slack variables and insert them as basic variables in the table `basicVariables`. Additionally to this task, in the two-phase method we need to add an artificial variable for every row, where no initial basic variable is available. To do so, we can use the table `basicVariables`, where the slack variables were inserted, to find the rows, where we need to add an artificial variable. For every row, where no basic variable is available yet, an artificial variable is added. When the first artificial variable is inserted in the relation, we know that the current problem does not have a basic feasible solution and it needs to execute the two-phase method. Therefore the tuples for an artificial objective function are inserted in the relation. In this case the function `getBasicVariables` returns `true`, which means that the two-phase method is needed and that the first phase inside the if-loop in the main function in Listing 4.8 is executed.

Listing 4.8: Main Function

```
CREATE OR REPLACE FUNCTION optimizelinearproblem
(IN problemtable character varying)
  RETURNS TABLE(col character varying, val double precision) AS
$$
DECLARE
   needFirstPhase BOOLEAN;
   state SimplexState;
BEGIN
   -- creating the table, which is returned at the end
   CREATE TABLE returnTable(col VARCHAR, val FLOAT);

   -- store the basic variables in the table basicVariables
   -- and add the necessary artificial variables to the problem,
   -- if one or more artificial variables are added,
   -- the two-phase method is executed
   SELECT getBasicVariables() INTO needFirstPhase;
```

```
  -- first phase of two-phase method
  IF needFirstPhase THEN
    -- compute relative cost coefficients
    -- in the artificial objective function row
    PERFORM transformObjectiveFunctionRow(1);
    -- apply Simplex method on the expanded problem,
    -- if optimal value is zero, state is 'optimal',
    -- otherwise the problem has no feasible solution
      state = optimize(1);
    IF state = 'optimal' THEN
      -- if basic variables still contain artificial variables,
      -- exchange them with non artificial variables
      PERFORM removeArtificialVariablesFromBasicVariables();
      -- remove the artifical variables and the
      -- artificial objective function row from the relation
      DELETE FROM problem_table AS a1
      WHERE a1.row = 'artObjective'
      OR a1.col LIKE 'artificial%';
      -- again compute relative cost coefficients,
      -- but this time in the objective function row
      PERFORM transformObjectiveFunctionRow(2);
    END IF;
  END IF;

  -- if we have a feasible solution, apply the Simplex method
  IF NOT needFirstPhase OR state = 'optimal' THEN
    state = optimize(2);
  END IF;

  -- fill the return table with the correct values
  fillReturnTable();

  -- return the results
  RETURN QUERY
    SELECT * FROM returnTable;
  DROP TABLE returnTable;
END;
$$ LANGUAGE PLPGSQL;
```

In the first phase, we need to minimize the artificial problem. Before we can do that, we need to transform the artificial objective function accordingly, so that it stores the relative cost coefficient. In the PLSQL implementation this transformation is done by calling the function `transformObjectiveFunctionRow`. In Listing 4.9 the main part of this function is presented. For each column we compute the value, which need to be subtracted from the objective function row and store these values in the table `valueToSubtract`. To calculate these values, we first need to get the values in the objective

function, which are belonging to a basic variable. Every row is subtracted as many times from the objective function that the value of its basic variable is zero in the objective function. So we multiply every value in the relation with the value in the objective function in the column, which belongs to the basic variable of the row of the current value. We then group the products of these terms by the column they belong to and sum these values up. The results of these additions are finally subtracted from the value in the objective function of the column they belong to.

Listing 4.9: Transform the objective function row

```
CREATE TABLE valueToSubtract(col VARCHAR, val FLOAT);
-- compute the value, which need to be subtracted
-- from the objective function for each column
INSERT INTO valueToSubtract
   SELECT a4.col, SUM(a3.val * a4.val)
   FROM
         (SELECT a1.row, a2.val FROM basicVariables AS a1
            INNER JOIN
            SELECT col, val FROM problem_table
            WHERE row = 'artObjective' AS a2
            ON a1.col = a2.col
         ) AS a3
         INNER JOIN
         problem_table AS a4
         ON a3.row = a4.row GROUP BY a4.col;

-- update the objective function accordant
-- to the previously computed values
UPDATE problem_table SET val = problem_table.val -
   (SELECT a1.val FROM valueToSubtract AS a1
      WHERE a1.col = problem_table.col)
   WHERE problem_table.row = 'artObjective';

DROP TABLE valueToSubtract;
```

Now that the artificial objective function is representing the relative cost coefficients, the artificial problem can be optimized. Like mentioned before, the function `optimize` is more or less the same, which is used for the initial example without the two-phase method, but has some small changes. The difference is that the optimizing step in the first phase refers to the artificial objective function, while the previous one referred to the real objective function. To know which objective function row should be considered, we add an `Integer` variable called `phase`. When `phase` is 1, we are in the first phase and the artificial objective function is considered and when it is 2, we are in the second phase and the real one is considered. The updated code is shown in Listing 4.10. Since for a feasible problem the optimal value of the artificial objective function need to be zero, another small difference to the old code is added. When we are in the first phase and no

pivot column is found, we need to check the optimal value if it is zero. If it is not, the problem has no feasible solution and we can return the `SimplexState not_feasible`. If it is zero, we return the `SimplexState optimal`, which in this case just means that we found a feasible solution and can continue with the second phase.

Listing 4.10: Optimize Function With Phases

```
CREATE OR REPLACE FUNCTION optimize(phase integer)
RETURNS SimplexState AS
$$
DECLARE
       pr VARCHAR;
       pc VARCHAR;
       optimalValue FLOAT;
BEGIN
   WHILE TRUE LOOP
      -- getting the pivot column or null if no value
      -- is negative in the objective function
       pc = getPivotColumn(phase);
      -- if no negative values,
      -- the optimum value is found
      IF pc IS NULL THEN
         IF phase = 1 THEN
            -- in first phase -> check if optimal value is zero
            SELECT val INTO optimalValue
            FROM temp_table
            WHERE row = 'temp' AND col = 'RHS';
            IF isEqualTo(optimalValue, 0) THEN
               RETURN 'optimal';
            ELSE
               RETURN 'not_feasible';
            END IF;
         ELSE
            RETURN 'optimal';
         END IF;
      END IF;
      -- getting the pivot row or null
      -- if there is no non-negative ratio
      pr = getPivotRow(pc);
      -- if no non-negative ratio
      -- the solution is unbounded
      IF pr IS NULL THEN
         RETURN 'unbounded';
      END IF;
      -- update the rows according to the pivot element
      Perform updateRows(phase, pc, pr);
      -- add the pivot column to the basic variables
```

```
      -- with the pivot row as row
      Perform updateBasicVariables(pc, pr);
   END LOOP;
END;
$$ LANGUAGE PLPGSQL;
```

Now the artificial problem is optimized and the first phase is finished. At this point we need to check if artificial variables are still contained in the basic variables. If there are, we need to call another Simplex step with the row, which contains the artificial variable as basic variable, as pivot row and a positive or negative value in this row as pivot column. Afterwards the artificial objective function row and the artificial variables can be deleted out, since they have no real meaning. This is done with a simple `DELETE` statement in Listing 4.8. At this state we have our initial problem with the updated values and with the current basic variables `Barley` and `Soy`. Similar to the first phase we need to compute the relative cost coefficients in the objective function. We again apply the same function `transformObjectiveFunctionRow` like before in Listing 4.9, but we replace the row name `artObjective` with `optimize` that now the real objective function row is transformed. To do so, we just give the phase as parameter and define the row to consider at the beginning of the function `transformObjectiveFunctionRow`. At this point that we have a feasible solution, we can execute the `optimize` function again to optimize the real problem. After doing this we can return the results. These values are returned the same way as in the first example, along with the optimum value. The returned table for the problem discussed in Section 2.3 is shown in Table 4.3. At the end, the used table are dropped and the result is shown.

Table 4.3.: Returned Table in PLSQL of the Second Example

col	val
soy	1.8
hay	0.4
barley	0
optimum_value	-2.2

5. Implement Simplex by extending the PostgreSQL kernel

Now that we have seen how the algorithm can be implemented in PLSQL, we want to start with the main part of the project, to implement the Simplex algorithm into the PostgreSQL kernel, so that the algorithm is executed by calling the correct query. A challenging part of this is that the table that represents a Linear Program needs to be updated after every iteration. Since PostgreSQL works in a pipeline manner, where an input relation is read and an output relation is created and given to the next plan node, the PostgreSQL kernel does not support iterations over a plan node, where the current iteration works with the output of the iteration before. Because of this, we need to find a method to run iterations over a relation. Before we tackle this problem, we go step by step through the process of extending the kernel. We need to edit the parser, the optimizer and the executor.

5.1. Parser

The parser checks if a given query has correct syntax by checking its rules. Hence we need to create appropriate rules for our Simplex algorithm. Before we create these rules we need to decide how the final query should look like and which information is needed by the executor. The only information we need to execute the Simplex algorithm, is to know which row is the objective function and which column is the right hand side.
In the PLSQL implementation the ordering of the input table does not matter, because SQL queries take over reordering the table when needed. However, in the C implementation inside the PostgreSQL kernel the ordering of the input table matters, because this implementation is low level and it controls the physical ordering of the tuples and reads the relation like it is ordered. We can control the ordering of the tuples during the execution, through that we can make sure that it stays always the same. Regarding this, we can use the ordering as a benefit. We decided to have our input table ordered by column and then by row. The ordering is done alphabetically, but the right hand side is the first column and the objective function the first row. Like this the first tuple of the relation belongs to the objective function and the right hand side and we can get the name of them by reading this tuple. The reason why we have decided to order the table by column and then by row will be discussed later in Section 5.3.1. The described ordering of the table is not done in the implementation and it is excepted, that the input table is already ordered like this.
Regarding such an input table the Simplex algorithm can be executed on that table

without any additional information. We just need to indicate in the query that we want to execute the Simplex algorithm on the input table. We add the keyword `SIMPLEX` at the end of the table name, on which we want to run the Simplex algorithm and we get the query shown in Listing 5.1.

Listing 5.1: Input query

```
SELECT * FROM input_table SIMPLEX;
```

Regarding this query the Simplex algorithm is executed over the table `input_table`. However, we want to extend this query with two optional parts of information. In Section 5.3 we create different versions of the C implementation and hence we want to let the user choose, which version he wants to execute. Besides choosing the version of the implementation, we want the user to be able to set the tolerance, which was already used in the PLSQL implementation as well. The tolerance is used to steer against small calculations error. Two values are handled as equal if their difference is smaller than the tolerance. Regarding these two extensions, the final query is shown in Listing 5.2.

Listing 5.2: Input query with version and tolerance

```
SELECT * FROM input_table SIMPLEX
simplex_version 1 simplex_tolerance '0.000001';
```

Considering this query, the first version of the Simplex implementation is executed over the table `input_table` with the tolerance set equal to 0.000001. Since the choosing of the version and the setting of the tolerance is optional, default values are set if one part of them is missing. The default values for both optional parts are the values used above in the query. We create a struct called `SimplexMethodInfo` to store the information, it is shown in Listing 5.3. Since we want to run an algorithm over a single table we can add a `simplexMethodInfo` attribute to the struct `RangeTblEntry` to transport the information to the executor. A `RangeTblEntry` is a node of the query parse tree that represents a relation and its attributes defined in the query. If the `SimplexMethodInfo` attribute in the `RangeTblEntry` is not `NULL` the Simplex algorithm will be executed.

Listing 5.3: SimplexMethodInfo

```
typedef struct SimplexMethodInfo
{
   NodeTag    type;
   char      *tolerance;
   int        simplexVersion;
} SimplexMethodInfo;
```

Now that we have decided how the final query should look like and we know how we want to store the information we can create according rules. Considering the query in Listing 5.2 our new rule will match the part after the `FROM` keyword and the input table. Listing 5.4 shows the parser rules that are invoked to parse this part of the query.

Listing 5.4: Invoked Rules of the Simplex Query

```
query: select_clause from_clause
from_clause: FROM table_ref
table_ref: relation_expr opt_simplex_method opt_alias_clause
```

First the query is split into the `SELECT` and the `FROM` clause. The `FROM` clause consists of the keyword `FROM` and the rule `table_ref`. The rule `table_ref` represents everything, where an alias clause can be attached, hence in our example the results of the Simplex algorithm. So we just add a new rule called `opt_simplex_method` as an opportunity in the rule `table_ref` along with a `relation_expr`, which represents the input table. The added part in the rule `table_ref` is shown in Listing 5.5. The rule `relation_expr` returns a `RangeTblEntry` and the `SimplexMethodInfo`, which is returned from the rule `opt_simplex_method`, is assigned to it as an attribute.

Listing 5.5: inserted part in the rule table_ref

```
table_ref: relation_expr opt_simplex_method opt_alias_clause
   {
      $$->simplex_info = (SimplexMethodInfo *) $2;
      $1-> alias = $3;
      $$ = (Node *) $1;
   }
   | ...
```

Listing 5.6 shows the added rule `opt_simplex_method`. It is either empty or consists of the word `SIMPLEX` along with additional optional information (`simplexVersion` and `tolerance`) in the rule `opt_simplex_information`. If it is empty, `SimplexMethodInfo` is `NULL` and the Simplex algorithm is not executed. Otherwise the values of the rule `opt_simplex_information` are assigned to `SimplexMethodInfo` or the default values if `opt_simplex_information` is empty. The Simplex algorithm will be executed in this case.

Listing 5.6: Main Parser Rule

```
opt_simplex_method:
SIMPLEX opt_simplex_information
{
    SimplexMethodInfo *n = $2;
    $$ = n;
}
| /* EMPTY */            { $$ = NULL; }
;
```

Now that we know how to write a query and how it will be parsed we can continue with the optimizer.

5.2. Optimizer

The main task of the optimizer is to find the cheapest way to execute the given query. Since there is only one possibility to execute the Simplex algorithm, namely our plan node, the optimizer has only one choice and needs to take the Simplex method path. However, we have mentioned in Section 5.1 that we want to create different versions of this algorithm and that the user decide, which one he want to pick to execute the algorithm. This decision should actually be part of the optimizer and it should pick the cheapest one for the current example. We need to make a cost analysis and calculate the I/O costs for both versions, to see if the costs are different. Since we need to know how the algorithm is processed in the executor, we skip the cost calculations here and come back to them after having discussed how the executor works. Section 6 considers the cost calculations and analyse them by executing different examples.

5.3. Executor

Now that we have discussed about the Parser and the Optimizer, we come to the main part of the implementation, the Executor. Before we can start with the real implementation of the Simplex algorithm we need to find a solution for the mentioned problem to loop over some updating commands. In the project the following 4 methods were investigated to do the mentioned task.

- Having one plan node, which iteratively updates the input relation and returns the final relation at the end.
- Having two plan nodes, where one executes one step of the Simplex algorithm and the other iteratively calls this node.
- Using a C array in the memory to store the relation and its intermediate results.
- Using so called tuplestores, which are relations we can build and modify by ourself in the C implementation.

To anticipate a bit, only the third and the forth method worked, but for completion the other methods are shortly described and explained why they did not work. For the third and the forth method a detailed explanation is given afterwards.
Probably the best method would be if we could just update the initial relation many times and then finally return the resulting relation. This was investigated in the first approach. The PostgreSQL kernel provides a method to update a tuple of a relation inside a plan node. More precisely, this function actually does not update a tuple inside a relation, it replaces an old tuple with a new one. The function takes as input three parameters. The first parameter is the relation, in which a tuple should be replaced. The second one is the pointer on the tuple, which should be replaced and the last one is the new tuple, which should be inserted at the position of the old tuple. This method works, if we want to update the table once during the execution of a plan node. However,

PostgreSQL does not allow to change the same tuple of a relation in a plan node twice, because the idea of PostgreSQL is that one plan node reads a relation and outputs a new one without changing its content many times. Additionally, the updated tuples are materialized and stored only at the end of the plan node execution when the relation is returned. So, if a tuple is updated using the function above and read again afterwards, the read tuple is the same, which it was before the update. The values in the relation are just updated when we call a new query, which is obviously too late. Out of these reasons this approach does not work to create a loop over updating commands.
The second method tried to work with two plan nodes, where one plan node executes one Simplex step and the other plan node iteratively calls this node with the relation returned by the previous Simplex step as input for the next step. Like this the plan node, which executes a Simplex step gets one input relation and can return the updated output relation, which fits with the idea of PostgreSQL. However, since the output relation of the Simplex step plan node is just returned by the plan node and hence it is never materialised, the next Simplex step can just read this output once when it is returned. Unluckily we need to read some values more than one time during a Simplex step and hence we can not apply this method to implement the Simplex algorithm. We see that we somehow need to materialise the intermediate results to be able to read them many times and hence we come to our last two methods.

5.3.1. Tuplestores

The third investigated method tries to materialise the relation and its intermediate results by storing every tuple in a C array in the memory. Like this we can access the tuples easily and also change them without a problem. However, when the relation is getting bigger and consisting of thousands of rows and columns, the amount of needed memory space grows extremely, possibly so far, that the needed space is too big to handle for a single computer.
To avoid increasing memory space regarding the size of the problems, we investigated a forth method, where we try to store as few things as possible in the memory. In the forth method we materialise the relation and its intermediate results using tuplestores. This means that we store the input table inside a tuplestore and afterwards execute every operation on the tuplestore and never touch the initial input relation again. Tuplestores are a generalized module for temporary tuple storage. At first a tuplestore actually does the same thing that we did in our third method right before, it just stores the tuples in an in-memory array. This approach leads to the same problem, that for big examples, the memory possibly would not have enough space for all the tuples. However, unlike the normal memory the tuplestores solves this problem by writing the tuples on a temporary file in the disk when the given amount of memory is not enough. Shortly explained, the tuplestore stores the tuples in an in-memory array until the given space limit is exceeded, then the tuples are written into a temporary file, from which the data can be read again when needed. To simplify the comparison of different tested examples later in Section 6, we only consider the case, where the tuplestore is written into the temporary file on the disk. To secure that this case happens nearly all the time, we pick the smallest possible

memory space limit (1 MB) for the tuplestore in memory, which means that the space limit is exceeded for nearly all examples. Therefore, in the following we assume that the tuplestore is written to the disk.
To scan through the relation a tuplestore uses an array of pointers. Initially the tuplestore has one read pointer, which points to the start of the table and which is the current active pointer. We can scan through the table by always getting the tuple the active pointer points to. When a tuple is gotten the active pointer increases its position and points to the next tuple. Therefore the active pointer always changes its state and points to the currently read tuple. To avoid expensive full scans, we can also add multiple pointers in the pointers array. When a new pointer is created, it points to the current position of the active pointer. Therefore many pointers to important positions of the tuplestore can be created during reading the table, which can be reused later. To get the tuples from the disk into the memory, a tuplestore provides a buffer storage inside the memory. This buffer storage has a space of 8 kilobytes. Regarding measurements, where we divided the total size of different linear optimization problem tables through the number of tuples, one tuple needs 40 bytes, hence the buffer can store around 200 tuples.
Algorithm 1 shows the process a tuplestore applies, when we want to get the tuple the active pointer points to. The tuplestore needs to store the tuple to get inside the buffer. The tuplestore first checks if this is already the case. This is not possible if the buffer is empty, which happens only when we have not read any tuple before, at the beginning of executing a Simplex step. So when the buffer is empty or just does not contain the tuple the active pointer points at, the tuplestore needs to get it from the disk. It computes the offset of the tuple to get and checks if it is contained in the next 200 tuples in the disk. In this case, the whole block of 200 tuples is copied into the buffer. Otherwise using the offset the tuplestore needs to compute in which block of tuples the tuple to get is contained, restore the file pointer on this block position and copy the whole block into the buffer. At this point the tuple to get is stored in the buffer and the tuplestore can get it and increase the active pointer. If the tuple is already in the buffer right from the beginning, this is directly done.

Algorithm 1 Process of get Tuple

```
if buffer_is_empty or !tuple_is_already_in_buffer then
  if !next_block_contains_tuple_to_get then
    restore_position_of_block_that_contains_tuple_to_get();
  end if
  read_block_into_buffer();
end if
get_tuple_and_increase_active_pointer();
```

Regarding that we sequentially scan a relation, in most cases the tuple to get next is already in the buffer and we do not need to access the disk. After 200 read tuples, we need to access the disk once, but we can just read the next block into the buffer, since

the tuple we want is the first one of this block. In a sequential scan we never need to search for the correct block in the disk. However, when we restore a pointer at a different position, the next tuple is maybe not in the current block in the buffer or in the next one on the disk and we need to search for the correct block in the disk and restore the file pointer. When we have a big table with thousands of tuples, the probability that a randomly restored position is in the current or the next block is very small and we can assume that we need to search for the correct position nearly every time.
Regarding restoring a pointers position, tuplestores provide two possibilities. The required position can be restored by copying the value of the required pointer into the current active pointer or by choosing the pointer with the required position as the new active pointer. With the method, where the position just gets copied, the copied pointer stays constant and can be reused again later. If the required pointer is chosen as the new active pointer, its pointing position changes when new tuples are read, hence with this method, the pointer is only temporary pointing at a position.
On the contrary to the PLSQL implementation, where no ordering could be expected inside a relation as long as we did not specify it, the ordering in the tuplestore is really critical, because we read one tuple after another as mentioned in Section 5.1. Pointers on important positions are just efficient, if we have a strict ordering and we know which tuple will be read next. To have a good access on different columns, we decided to have an input table ordered by columns and then by rows. We create pointers on the start of each column, so that we can jump to every column we want to without reading unnecessary tuples. Since the relative cost coefficient row is really critical in the Simplex algorithm and need to be accessed many times, we want pointers on this row as well. Instead of creating new pointers, we can use the same pointers by ordering the columns, so that the first tuple of every column belongs to the relative cost coefficients row. By doing this, the pointers, which are pointing on the first tuple of each column, are pointing on the relative cost coefficients row at the same time. They can be used to find the pivot column much more efficient. We will come to that in more detail later.
In Section 5.1 we also expected the right hand side column to be the first column, so that we can differ it from the other columns. Table 5.1 shows a part of the initial tuplestore, the first column in the table indicates the pointers, which are pointing on the corresponding column. Since pointer 0 is the active pointer, the tuple (optimize, RHS, 0) is the tuple, which will be read next. Table 5.2 shows the same table after this tuple is read. Pointer 0 is now pointing at the tuple (CP, RHS, 2). If we want to jump to the nth column, we can just copy the data of the pointer at position n to the active pointer at position 0.
Besides just getting the tuple into the memory we need to apply an additional step to read the information of a tuple of the tuplestore, because the tuples are stored as binary values. Hence to get the information of a tuple we need to deserialize the binary value first. In the other way, if we want to put a new value on the tuplestore, we need to first serialize the data of the new tuple that it is a binary value again and can be put on the tuplestore. It is possible that we do not really care about the information of a tuple and never need to deserialize it, because we only want to move the active pointer one position forward or to put the current tuple on another tuplestore. We use the following

Table 5.1.: Table With Pointers

pointer	row	col	val
0, 1	optimize	RHS	0
	CP	RHS	2
	LYS	RHS	4
2	optimize	Barley	12
	CP	Barley	3
	LYS	Barley	1
3	optimize	Hay	1
	CP	Hay	1
	LYS	Hay	3
4	...	...	...

Table 5.2.: Table With Pointers After a Tuple is Read

pointer	row	col	val
1	optimize	RHS	0
0	CP	RHS	2
	price	RHS	4
2	optimize	Barley	12
	CP	Barley	3
	price	Barley	1
3	optimize	Hay	1
	CP	Hay	1
	price	Hay	3
4	...	...	...

5 basic operations in connection with the tuplestore:

- get the next tuple (sequential get)
- get a random tuple (random get)
- deserialize a tuple
- serialize a tuple
- put a tuple on a tuplestore

Since deserializing a tuple only makes sense when a new tuple is gotten from the tuplestore before, we combine some of the basic operations for better understanding of the code and the cost calculations. The actual operations, which are used in the implementation are the following:

- sequential read (sequential get and deserialize a tuple)
- random read (random get and deserialize a tuple)
- write (serialize and put a tuple)
- sequential get
- put

The last two operations are still the basic operations, because they can appear individually as well. Regarding these basic operations, we will use the following functions in the code examples.

- void skip_tuples(Tuplestore tupstore, int tuplesToSkip);
  - Gets `tuplesToSkip` tuples in `tupstore`, to move the pointer forward.
- TupleSlot get_tuple(Tuplestore tuplestore);
  - Gets the next tuple from `tupstore`. Returns the tuple as binary value (`TupleSlot`).
- SimplexMatrixElement read_tuple(Tuplestore tupstore);
  - Gets the next tuple from `tupstore` and deserializes its content. Returns the deserialized tuple (`SimplexMatrixElement`).
- void put_tuple(Tuplestore tupstore, TupleSlot tupleSlot);
  - Puts (appends) a `TupleSlot` on to `tupstore`.
- void write_tuple(Tuplestore tuplestore, SimplexMatrixElement matrixElement);
  - Serialises and puts a `SimplexMatrixElement` on to `tupstore`.
- void create_pointer(Tuplestore tupstore);
  - Creates a constant pointer on the current position of pointer 0 in `tupstore`.
- void copy_column_pointer_in_active(Tuplestore tupstore, int columnToCopy);
  - Copies the pointer on the column `columnToCopy` in the active pointer in `tupstore`. Through that the column `columnToCopy` is restored.
- void create_temp_pointer(Tuplestore tupstore, int columnToRead);
  - Creates a temporary pointer on the column `columnToRead` and picks it as active pointer in `tupstore`.

- void select_active_pointer(Tuplestore tupstore, int pointerPosition);
  - Selects the pointer at position `pointerPosition` as active pointer in `tupstore`. Is only used for the temporary pointers and the pointer 0. For example if the variable `pointerPosition` is one, the first temporary pointer is the new active pointer. If a temporary pointer is the current active pointer and it is not needed any more, we need to select pointer 0 as active pointer again by executing this function with $pointerPosition = 0$.

Unluckily it is not possible to update tuples inside a tuplestore, it is only possible to insert tuples at the end (append). This makes updating the intermediate relations more complicated. However, it is possible to create as many tuplestores as wanted, therefore it is possible to create a new tuplestore every time we want to change something inside it. We can just put every tuple from the old one to the new one along with the wanted changes. This is quite bad, because in some cases we only need to update a part of the table and like this we always need to get and put the non updated tuples that we have these tuples on the new tuplestore as well. We will come to this again later in the explanation of the implementation. Additionally we always need to create a new tuplestore and new pointers on the wanted positions, which are expensive operations. We create one tuplestore for every iteration in the Simplex algorithm.

5.3.2. Two versions of the Simplex algorithm

Before we come to the implementation of the Simplex algorithm we want to consider another example of a problem table, which is presented in Table 5.3.

Table 5.3.: Problem Table With Many Zeros

Constraint ID	RHS	var1	var2	var3	var4	s1	s2	s3
optimize	0	-3	-5	-2	-2	0	0	0
CP	1	0	0	4	0	1	0	0
DEP	2	2	0	1	0	0	1	0
LYS	6	0	3	0	2	0	0	1

We can see that there are many zero values inside this problem table and probably it is not really efficient to store them all in the tuplestore. Out of this reason we create two versions of the PostgreSQL implementation, one which stores all the zero values inside the tuplestore and another one which stores only the non-zero values. Because of not storing the zero values we do not know the exact position of every tuple, because some rows could possibly be skipped out. To reduce this loose of control, we still store every tuple, even the zero value tuples, of the first column (the right hand side), to be able to read all existing rows once and store their position, as well as every tuple of the relative cost coefficient rows, to keep the pointers on these tuples. Table 5.4 shows a part of the new problem table with the according pointers for both versions, once with all zero values and once without.

Table 5.4.: Table With Pointers for Version With and Without Zero Values

pointer	row	col	val
0, 1	optimize	RHS	0
	CP	RHS	1
	DEP	RHS	2
	LYS	RHS	6
2	optimize	var1	-3
	CP	var1	0
	DEP	var1	2
	LYS	var1	0
3	optimize	var2	-5
	CP	var2	0
	DEP	var2	0
	LYS	var2	3
4	...	...	...

pointer	row	col	val
0, 1	optimize	RHS	0
	CP	RHS	1
	DEP	RHS	2
	LYS	RHS	6
2	optimize	var1	-3
	DEP	var1	2
3	optimize	var2	-5
	LYS	var2	3
4	...	...	...

Regarding the efficiency of these two versions, we will run different examples in Section 6 and compare the running times of them for both versions. To confirm these experiments we will compare the amount of sequential read, random read, write, get and put operations, which are executed during the Simplex method. In the following we call the version with zero values the dense version and the one without zero values the sparse version. To calculate the amount of operations executed we use the parameters `m` as the number of rows, `n` as the number of columns, `nz` as the percentage of non-zero values in total and `pr` as the percentage of non-zero values in the pivot row. Beside these two versions also a version using a C array in memory to materialise the intermediate results, which was our third approach to solve the updating problem mentioned earlier in this section, will be studied. However, because this implementation is really simple it will not be described in the following. It will be used just in Section 6 to compare the different implementations regarding their efficiency.

5.3.3. Simplex algorithm in PostgreSQL

Now that we have found a solution for updating the problem table and discussed about different versions of the implementation, we can start to implement the algorithm in the PostgreSQL kernel. We again start with an example with only the second phase and add the additional steps for the first phase afterwards. Regarding the two versions of the implementation using tuplestores to materialise the intermediate results, we will mainly explain and visualize the dense version and then mention the differences between the two methods afterwards. We work with the state node, which is presented in Listing 5.7. The first field `ss` is of the type `ScanState` and stores everything regarding the input table. With the help of this field we can scan the input table and store it inside the tuplestore. Since the Simplex algorithm first needs to be executed before any tuple can

be returned and afterwards the tuples can just be returned without doing something more, it can be splitted into two states. The first is the execution of the Simplex algorithm and the second is the returning of the result tuples. To know the stage at which the execution is, we have the variable `currentStatus`. It stores the result type of the algorithm similar to the `SimplexState` in the PLSQL implementation. It can have the value `RUNNING`, `OPTIMAL`, `NOT_FEASIBLE` or `UNBOUNDED`. When the status is `RUNNING`, we need to execute at least one more Simplex step and the solution can still be further improved. `OPTIMAL` means that the optimal solution is found, while `NOT_FEASIBLE` and `UNBOUNDED` are standing for a problem without a solution and a problem with an unbounded solution respectively. At the beginning of the executor the `SimplexState` obviously is `RUNNING`. In the fields `pc` and `pr`, the position of the pivot column and of the pivot row is stored. The attribute `pe` stores the value of the pivot element. In the integers `numberOfColumns` and `numberOfRows` the number of columns and rows are stored. In the two-phase method we also store the number of the column, where the first artificial variable appears in the attribute `startOfArtificials`. By knowing this, we can make for loops over all columns but the artificial variables. Since this is the only attribute, which is added for the two-phase method, it is already mentioned here, that it is not necessary to repeat the whole state later, when we consider the two-phase method. The next variable `numberOfReturnedTuples` is for the returning part of the code. It stores the number of tuples, which have already been returned, that we know which tuple needs to be returned next. The variable `tolerance` is similar to the tolerance already seen in the PLSQL implementation. Two values are considered to be the same, if their difference is smaller than the variable `tolerance`. In the following code examples, for simplicity comparisons of two values are done with the usual $>$,$<$ and $==$ signs, while in the real code the tolerance is used. The boolean variables `useCArray` and `withZeroCells` contain the information, which implementation needs to be executed. In the next two variables the tuplestores are stored included their pointers. `tupstore1` stores the initial table and the updated values from it are stored in `tupstore2`. After every Simplex step `tupstore1` is freed and `tupstore2` is copied in `tupstore1`. Like this we can start with `tupstore1` as initial table again like before. The next attribute `basicVariables` stores the current basic variables of the table. For every row but the objective function row there is a basic variable. For example `basicVariables[0]` stores the position of the basic variable in the first row, along with the value of it in the right hand side. The attribute `namesOfRows` stores the name of the rows. It is only needed for the sparse version, because there we need to be able to recognize tuples according to their row names, while in the dense version we can recognize a tuple only with its position in the table. The last attribute `matrix` represents the C array, which is used to materialise the intermediate relations. It is an array of `SimplexMatrixElement` nodes. A `SimplexMatrixElement` stores the information of one tuple. It has the attributes `row`, `col` and `val`. Of course the array is only used in the implementation, which materialise the relation like this.

Listing 5.7: SimplexMethodState

```
typedef struct SimplexMethodState
{
   ScanState                  ss; /* its first field is NodeTag */

   SimplexStatus           currentStatus;
   int                     pc;
   char                    *pr;
   int                     prNumber;
   double                  pe;
   int                     numberOfColumns;
   int                     numberOfRows;
   int                     startOfArtificials;
   int                     numberOfReturnedTuples;
   double                  tolerance;
   bool                    useCArray;
   bool                    withZeroCells;

   Tuplestorestate         *tupstore1;
   Tuplestorestate         *tupstore2;

   BasicVariableElement    **basicVariables;

   // only needed for the sparse version
   char                    **namesOfRows;

   // only needed for the version using a C array in memory
   SimplexMatrixElement    **matrix;

} SimplexMethodState;
```

Now that we have defined the state node we can start with the implementation of the Simplex algorithm. The `SimplexMethodState` is initialized in the `ExecInitSimplexMethod` method. In Listing 5.8 the most important parts of this method are presented. The tolerance can be copied from the plan node `SimplexMethod`. As mentioned before, the value of the attribute `currentState` needs to be `RUNNING` at the beginning, because we still need to execute the Simplex algorithm. The columns and rows counters (`numberOfColumns` and `numberOfRows`) are initialized here with zero. Since we did not return a tuple yet, `numberOfReturnedTuples` is zero as well. The booleans `useCArray` and `withZeroCell` can be deduced from the variable `simplexVersion` in our plannode. If the variable `simplexVersion` is bigger than 2, the implementation using a C array in the memory is used. If it is smaller than 2 the implementation using a tuplestore with zero value cells and if it is equal to 2 the implementation using a tuplestore without zero value cells is used. At the end the first tuplestore is initialized and assigned to `tupstore1`.

Listing 5.8: ExecInitSimplexMethod

```
SimplexMethodState *
ExecInitSimplexMethod(SimplexMethod *node,
                                  EState *estate, int eflags)
{
      SimplexMethodState *simplexstate;
      simplexstate = makeNode(SimplexMethodState);

      simplexstate->tolerance = node->simplex_info->tolerance;
      simplexstate->currentStatus = RUNNING;
      simplexstate->numberOfColumns = 0;
      simplexstate->numberOfRows = 0;
      simplexstate->numberOfReturnedTuples = 0;


      if(node->simplex_info->simplexVersion > 2){
         simplexstate->useCArray = true;
      }else{
         simplexstate->useCArray = false;
         if(node->simplex_info->simplexVersion < 2){
            simplexstate->withZeroCells = true;
         }else{ // simplexVersion == 2
            simplexstate->withZeroCells = false;
         }
      }
      simplexstate->tupstore1 = begin_tuplesstore();

      ...

      return simplexstate;
}
```

After we initialized our state node we can start with the actual algorithm in the function `ExecSimplexMethod`. The code of it is shown in Listing 5.9. This function only works for simple problems that have an initial feasible solution and an extended version will be shown in Section 5.3.4, where the two-phase method is handled. Like mentioned, the Simplex method needs to be processed only in the first invocation of the `ExecSimplexMethod`. To secure that, the according code is inside an if-clause, which checks if the `currentStatus` is `RUNNING`. This is only true at the first invocation. Inside the if-clause the method `storeInTupleStore` is called at the beginning. This method copies the tuples from the input relation `node->ss.ss_currentRelation` on the state node to the tuplestore `node->tupstore1`, which is also on the state node. To do so, it reads every tuple in the relation and stores it in the tuplestore. While doing this, it also counts the number of rows and of columns and stores the number in the state node. For the sparse version it additionally fills the array `namesOfRows` with the name of the rows by reading

the first column. Since this method also adds artificial variables if needed, it is discussed more deeply in Section 5.3.4, where we discuss about the two-phase method.

Listing 5.9: ExecSimplexMethod

```
TupleTableSlot *
ExecSimplexMethod(SimplexMethodState *node)
{
   if(node->currentStatus == RUNNING){
      // storing the input table in a tuplestore
      storeInTupleStore(node);

      // apply Simplex method
      while(node->currentStatus == RUNNING){
         node->currentStatus = simplexStep(node);
      }
      getOutput(node);
   }
   // return the correct values
   return returnATuple(node);
}
```

After the input table is copied on to the tuplestore we can start searching for the optimal solution and iteratively execute Simplex steps. Hence the function `simplexStep` is called inside a while-loop as long as `currentStatus` is equal to `RUNNING`. To know when the value of `currentStatus` changes, we need to look at the code of the function `simplexStep`, it is shown in Listing 5.10. Every step of it is described in the following.

Listing 5.10: Code of the function simplexStep

```
SimplexStatus simplexStep(SimplexMethodState *node){
   initializePointers(node);
   getPivotColumn(node);
   if(node->pc == -1){
      return OPTIMAL;
   }
   getPivotRow(node);
   if(node->pr == -1){
      return UNBOUNDED;
   }

   node->tupstore2 = begin_tuplestore();

   for(i = 0; i < node->numberOfColumns; i++){
      if(i == node->pc){
         getAndUpdatePivotCol(node);
      }else{
         updateCol(node, i);
```

```
      }
   }
   updateBasicVariables(node);
   resetTupleStores(node);
   return RUNNING;
}
```

At the start of this function we come to the first usage of the pointers of the tuplestore. As mentioned as a disadvantage, we need to create the pointers on the important positions in every new tuplestore again. We decided to create a pointer on the start of each column, which are the tuples belonging to the relative cost coefficients row at the same time. These pointers are created once per execution of the function `simplexStep`, because each time the table is updated, a new tuplestore is created, which needs new pointers. The function `initializePointers`, which creates the pointers for the dense version, is shown in Listing 5.11. When a new pointer is created, it always copies the value of the pointer at position zero. Therefore we scan through the table using the pointer 0 and create the new pointers on the corresponding positions. The pointer 0 is created along with the tuplestore and it is the active pointer at the beginning. Since our first pointer should point on the first column and hence at the start of the table, the first pointer needs to be created right at the beginning of the table. To create the pointers on the other columns, we always skip as many tuples as the variable `numberOfRows`, because one column has one tuple per row, and then create a new pointer. We do that until we have created one pointer for each column. By knowing the exact ordering of the table and the exact position of the beginning of each column, we never need to deserialize a tuple and we can save a lot of time. The only cost we get, is the cost of skipping tuples to find the correct positions. Skipping tuples actually just means that we get every tuple we skip once, to move the active pointer forward in the relation. We need to skip `n` columns, which have `m` tuples each, which makes a total of $m \cdot n$ tuples. Hence we need to get every tuple of the table once. Unluckily, in the sparse version this exact positioning does not exist, because some tuples are missing. Therefore we need to deserialize every tuple we get and check if it belongs to a new column. In the sparse version, we need to read (get and deserialize) every tuple of the table once. Since the sparse table consists of $m \cdot n \cdot nz$ tuples, we need to process this number of tuples. In both versions we can sequentially get the tuples.

Listing 5.11: Initializing of the pointers

```
void initializePointers(SimplexMethodState *node){
   // constant pointer
   // -> numberOfPointers should be 1 after first allocation
   numberOfPointers = create_pointer(node->tupstore1);
   while(numberOfPointers < node->numberOfColumns){
      skip_tuples(node->tupstore1, node->numberOfRows);
      // constant pointer
      numberOfPointers = create_pointer(node->tupstore1);
   }
```

```
}
```

Now that we have initialized the pointers, we can start with the Simplex algorithm and look for a pivot column. Listing 5.12 shows the code that finds the pivot column. Considering that the sparse version stores the whole relative cost coefficients row, this method is the same for both versions. Since we need to find the column with the most negative value in the relative cost coefficients row, we store the current minimum in the variable `min` and the position of the column with the current minimum in the variable `pc`. For the case that no value is negative and no column can serve as pivot column, we assign -1 to `pc` and 0 to `min` at the beginning. For every column we restore the position by copying the pointer of the column into the active pointer at position 0. We check if the relative cost coefficient in the current column is smaller than the current minimum `min`, if it is, we replace the value in `min` with the new value and `pc` with the current column. Since the tuple, which belongs to the relative cost coefficient, is the first tuple in every column we just read one tuple per column. Hence we need to read n tuples to get the pivot column. Since we need to restore the position before every reading of a tuple, we first need to search for the position of the tuple in the tuplestore and hence we need to process random reads. Since we never change the active pointer, the pointers on the starts of the columns stay constant. If there is no negative value in the relative cost coefficients row, `node->pc` is assigned to -1 and the current solution is optimal. The function `optimizeStep` can return `OPTIMAL` as `SimplexStatus` and the function `ExecSimplexMethod` can start to return the final values.

Listing 5.12: Get Pivot Column

```
void getPivotColumn(SimplexMethodState *node)
{
   double min = 0;
   int pc = -1;

   // for each column but the RHS
   for(i = 1; i<node->numberOfColumns; i++){
      copy_column_pointer_in_active(node->tupstore1, currentCol);
      matrixElement = read_tuple(node->tupstore1);
      if(matrixElement.val < min){
         min = matrixElement->val;
         pc = i;
      }
   }

   node->pc = pc;
}
```

If a pivot column could be found, we need to look for the pivot row. Listing 5.13 shows the corresponding code. We need variables for the current minimum ratio `minRatio`, the position of the row with the current minimum ratio `pr`, as well as the current pivot

element pe. We assign -1 to pr for the case that no pivot row can be found. At this point we need to use temporary pointers for the first time, because we need to read two columns in parallel, the pivot column as well as the right hand side column. We create one temporary pointer for both columns and skip the first tuple for each pointer, because this tuple belongs to the cost coefficients row and this row can not be chosen as the pivot row. The temporary pointers are chosen as the active pointer during this, hence their position changed and we now have a pointer on the second tuple of the pivot column as well as one on the second tuple of the right hand side column. For each row, we read one tuple in the pivot column and one tuple in the right hand side column. Like this we scan both columns exactly once.

Listing 5.13: Get Pivot Row

```
void getPivotRow(SimplexMethodState *node){

   int pr = -1;

   // creates a temporary pointer on the pivot column
   // and selects it as active pointer
   create_temp_pointer(node->tupstore1, node->pc);
   skip_tuples(node->tupstore1, 1);
   // creates a temporary pointer on the RHS column
   // and selects it as active pointer
   create_temp_pointer(node->tupstore1, 1);
   skip_tuples(node->tupstore1, 1);

   // for each row, but the objective function
   for(i = 0; i<numberOfRows - 1; i++){
      // selects the temporary pointer on the pivot column as active pointer
      select_active_pointer(node->tupstore1, 1);
      matrixElementPC = read_tuple(node->tupstore1);
      // selects the temporary pointer on the RHS column as active pointer
      select_active_pointer(node->tupstore1, 2);
      matrixElementRHS = read_tuple(node->tupstore1);
      if(matrixElementPC.val > 0){
         currentRatio = matrixElementRHS.val / matrixElementPC.val;
         if(currentRatio >= 0 &&
         (currentRatio < minRatio || pr == -1)){
            minRatio = currentRatio;
            pe = matrixElementPC.val;
            pr = currentRow;
         }
      }
   }

   select_active_pointer(node->tupstore1, 0);
   node->pe = pe;
```

```
   node->pr = pr;
}
```

By choosing the temporary pointers as the current active pointer, their position changes when a new tuple is read and we can read the columns in parallel. Table 5.5 shows this process with focus on the pointers. Since 'Barley' is the pivot column, we only care about this column and the right hand side. The temporary pointers are named p1 and p2. The first table shows the initial state of the loop, where we just skipped the first tuple of both rows. After we entered the loop we select p1 as active pointer and we read the next tuple, which is the next one in the pivot column. p1 moves one tuple forwards and the second table shows the current stage of the pointers. We read the value of the same row in the right hand side column by choosing p2 as active pointer. p2 moves one tuple forwards and the third table shows the current stage of the pointers. At this point we have the two necessary values to calculate the ratio of the 'CP' row and we divide the value of the right hand side column through the value of the pivot column. Before we do this, we need to check if the value in the pivot column is positive. If it is not, we can skip this row, because it can not be the pivot row, otherwise we can calculate the ratio. If the calculated ratio is not negative and smaller than the current minimal ratio in `minRatio` or `pr` is still -1 and hence it is the first ratio, we replace the value in `minRatio` with the new ratio and the current pivot row and pivot element with the values of the new row. We repeat these steps for all rows and finally store the pivot row and the pivot element in our state node. The forth and fifth table in Table 5.5 show the stage of the pointer during reading the values of the second row. All together we need to read m tuples in the pivot column and m tuples in the right hand side column. Since we jump from column to column, we need to restore the active pointer every time we read a tuple, hence we have random reads. In total we need to randomly read $2 \cdot m$ tuples to find the pivot row. In the sparse version the processing is quite similar. The only difference is that in the pivot column it is possible that some zero value tuples are missing, but not in the RHS column. To check if any tuple is missing we use the array `namesOfRows`, which stores the name of the rows. We compare the row we are reading now and the row we have read before. The difference of the indexes of these rows in the array `namesOfRows` subtracted by one is the number of zero value tuples that are missing before the currently read tuple. Since the value in the pivot column need to be positive in the pivot row, these rows can not serve as the pivot row and we can skip these rows in the right hand side. We can skip these tuples without deserializing them, because we know that even the zero value tuples are contained in this column and hence the ordering is fixed. In this version we need to randomly read only $nz \cdot m$ tuples in the pivot column and the same amount in the right hand side, which is in total $2 \cdot nz \cdot m$. Beside this we need to skip $(1 - nz) \cdot m$ tuples in the right hand side. In both versions `node->pr` is assigned to -1, if there is no positive value in the pivot column and the solution is unbounded. The function `optimizeStep` can return `UNBOUNDED` as `SimplexStatus` and the function `ExecSimplexMethod` can start to return the final values, which are unbounded in this case. At the end of the function, the pointer at position zero is chosen as the active pointer again.

Table 5.5.: Pointers During Determination of the Pivot row

pointer	row	col	val
0, 1	optimize	RHS	0
p2	CP	RHS	2
	LYS	RHS	4
2	optimize	Barley	12
p1	CP	Barley	3
	LYS	Barley	1
3	...	...	...

pointer	row	col	val
0, 1	optimize	RHS	0
p2	CP	RHS	2
	LYS	RHS	4
2	optimize	Barley	12
	CP	Barley	3
p1	LYS	Barley	1
3	...	...	...

pointer	row	col	val
0, 1	optimize	RHS	0
	CP	RHS	2
p2	LYS	RHS	4
2	optimize	Barley	12
	CP	Barley	3
p1	LYS	Barley	1
p1, 3	...	...	...

pointer	row	col	val
0, 1	optimize	RHS	0
	CP	RHS	2
	LYS	RHS	4
p2, 2	optimize	Barley	12
	CP	Barley	3
	LYS	Barley	1
p1, 3	...	...	...

Now that we have found the pivot column and the pivot row, we can start to update the values in the tuplestore. First we initialize the second tuplestore `tupstore2`, in which the updated values are stored. The updating of the pivot column and the updating of the normal columns is separated in two functions, one to update the pivot column and one to update a non-pivot column. To keep the ordering of the columns in the table exactly the same as it was before the update, we loop through the columns and update the columns one by one according to whether it is the pivot column or not. First we will consider the function to update the pivot column and afterwards we investigate the updating of a normal column. The code for updating the pivot column is shown in Listing 5.14. First we copy the pointer on the pivot column in the pointer 0, which is the active pointer. For every row in the pivot column we read the corresponding tuple, update the value and push the updated tuple on the new tuplestore `tupstore2`. If the row is the pivot row, the value needs to be one and otherwise it needs to be zero. Since we do this for every row once, we need to read and write m tuples. The updating of the pivot column in the sparse version is much easier. Since all values in the pivot column but the one, which belongs to the pivot row, are zero, we just need to store the tuple of

the pivot row and the one of the relative cost coefficients row. In total we need to write two tuples and read a single one, to keep the correct name of the pivot column.

Listing 5.14: Update pivot column

```
void getAndUpdatePivotCol(SimplexMethodState *node){

   copy_column_pointer_in_active(node->tupstore1, node->pc);

   for(i = 0; i<node->numberOfRows; i++){
         matrixElement = read_tuple(node->tupstore1);
         if( i == node->pr){
            matrixElement.val = 1;
         }else{
            matrixElement.val = 0;
         }
         write_tuple(node->tupstore2, matrixElement);
   }
}
```

Next to update the pivot column we also need to update the other columns. Listing 5.15 shows the corresponding function. This function is called for every column but the pivot column once and hence the position of the current column to update is a parameter of the function. To update a tuple in the Simplex algorithm we need to know the current value of the tuple, the value of the updated pivot row in the current column and the value in the pivot column in the current row. Since we update column by column the value in the pivot row is the same for all the tuples we want to update in one function call. Therefore we read and store this value in the variable `valueInPivotRow` at the beginning of the function. To do so, we copy the position of the current column in the active pointer, skip as many tuples that the next tuple is the pivot row and then read and store the value of the next tuple. Because we need the updated value of the pivot row, we need to divide the read value through the value of the pivot element, which is stored in our state node. Since the value in a tuple gets subtracted by the value in the pivot row multiplied by the value in the pivot column, the values of the tuples in the current column do not change during this Simplex step if the value in the pivot row is zero. In this case, we can just get all tuples of the current column without deserializing and put them on the new tuplestore without serializing. In the case, where the value in the pivot row is not zero, we need to read the column to update and the pivot column in parallel, similarly like we did earlier to find the pivot row. We create temporary pointers on both of the columns and then alternately read a tuple of both columns by selecting the temporary pointers as active pointers. For every row but the pivot row we calculate the new value of the according tuple using the corresponding pair of values and put the updated tuple on the tuplestore `tupstore2`. In the pivot row we can just replace the old value with the value in the variable `valueInPivotRow`.
To get the value of the variable `valueInPivotRow` we need to skip in average $1/2 \cdot m$ tuples and read one tuple. Assuming that this value is not zero, we need to read $2 \cdot m$

tuples and write m tuples to update the current column. Equally as to get the pivot row, we jump from column to column to read the next tuple, so we have again only random reads in this case. Regarding that the variable `valueInPivotRow` is zero we only need to get and put the tuples of the current column without deserializing or serializing a tuple, because the values stay the same. The probability that the variable `valueInPivotRow` is not zero should be similar to the percentage of non-zero values `nz`. However, experiments in Section 6 showed, that the percentage of non-zero values in the pivot row and the percentage of non-zero values in the whole table can differ extremely. Because of this, we want to have a different variable for this percentage. In the following the parameter `pr` stands for the probability that the variable `valueInPivotRow` is not equal to zero. Regarding this we need to process in average around $2 \cdot pr \cdot m$ random reads, $pr \cdot m$ writes, $(1.5 - pr) \cdot m$ single sequential gets and $(1 - pr) \cdot m$ single puts to update one column. Since we need to update every column once, these values can be multiplied by n to update the whole table.

Listing 5.15: Update normal column

```
void updateCol(SimplexMethodState *node, int colToUpdate){

   // get the value in the pivot row of the current column
   copy_column_pointer_in_active(node->tupstore1, colToUpdate);
   skip_tuples(node->tupstore1, node->pr);
   matrixElement = read_tuple(node->tupstore1);
   valueInPivotRow = matrixElement.val / node-> pe;
   if(valueInPivotRow != 0){
      // update the current column
      create_temp_pointer(node->tupstore1, colToUpdate);
      create_temp_pointer(node->tupstore1, node->pc);
      for(i = 0; i<node->numberOfRows; i++){
         select_active_pointer(node->tupstore1, 1);
         matrixElementCC = read_tuple(node->tupstore1);
         select_active_pointer(node->tupstore1, 2);
         matrixElementPC = read_tuple(node->tupstore1);
         if(currentRow == node->pr){
            matrixElementCC.val = valueInPivotRow;
         }else{
            matrixElementCC.val = matrixElementCC.val -
                                  matrixElementPC.val * valueInPivotRow;
         }
         write_tuple(node->tupstore2, matrixElementCC);
      }
      select_active_pointer(node->tupstore1, 0);
   }else{
      // get and put the current column
      copy_column_pointer_in_active(node->tupstore1, colToUpdate);
      for(i = 0; i<node->numberOfRows; i++){
```

```
        tupleSlot = get_tuple(node->tupstore1);
        put_tuple(node->tupstore2, tupleSlot);
      }
   }
}
```

In the sparse version the proceeding is again really similar, but there are some differences. At the beginning we also need to find out the value of the variable `valueInPivotRow`. However, in this version we can not just skip some tuples to find the value we need. We need to read every tuple in the current column, to check if it is the one we need, until we found the correct one. This makes an average of $1/2 \cdot m \cdot nz$ sequential reads to find the correct value. If the needed value is not found in the current column, it is zero and we can just get and put the current column without changes as done before. However, this time we need to deserialize every tuple, to check if we are still in the correct column. Otherwise we need to read the pivot column and the current column in parallel similarly as done before. In this version some values are missing and we need to check the row of the read tuples, to update the tuples correctly, but this does not affect the number of read tuples. Summarized we need in average $1/2 \cdot m \cdot nz$ sequential reads to find the value in pivot row and in average $2 \cdot pr \cdot m \cdot nz$ random reads, $pr \cdot m \cdot nz$ writes, $(1 - pr) \cdot m \cdot nz$ sequential reads and $(1 - pr) \cdot m\dot{n}z$ single puts to update one column when the value in the pivot row is already known. These costs combined multiplied by the number of columns, this makes a total of $(1.5 - pr) \cdot m \cdot n \cdot nz$ sequential reads, $2 \cdot pr \cdot m \cdot n \cdot nz$ random reads, $pr \cdot m \cdot n \cdot nz$ writes and $(1 - pr) \cdot m \cdot n \cdot nz$ single puts to update the whole table.
Considering these cost calculations it needs to be mentioned that the non-zero values are not evenly distributed in the table. The columns of basic variables have much less non-zero values than the rest of the table. Since most of the basic variables (all but one) have a zero value in the pivot row, the columns, where the value in the pivot row is zero, have in average less non-zero values than the columns, where it is not zero. Table 5.6 shows an example, which has `nz` and `pr` equal to 60%. The columns, where the pivot row is zero are written in italics. We can see that in the part, which concludes these columns, the percentage of non-zero values(objective function included) is only 45%, while in the other part the percentage of non-zero values is 70%. So the percentage of non-zero values is higher in the part, where we need to update the tuples. Therefore more tuples than we estimated before need to be updated and less can just be read and put on the new tuplestore, hence we need to process more operations than calculated before. Such a difference in the percentage of non-zero values between these two parts theoretically occurs in every example. However, the influence of this difference of the percentage is minor regarding the total cost to update the columns and we will not consider this more deeply to avoid unnecessary complexity. Hence we let our cost calculation for the updating of the columns how it is, but we need to have in mind that it is probably a bit optimistic.
For both versions we mentioned, that we just read and write the whole column, if the variable `valueInPivotRow` is equal to zero. This can happen very often if we have a

Table 5.6.: Distribution of Non-Zero Values

constraint	pivot var	var2	var3	var4	var5	*var6*	*var7*	*var8*	*var9*	RHS
row1	0	3	0	2	0	*4*	*1*	*0*	*0*	2
row2	4	0	0	0	0	*0*	*0*	*1*	*0*	4
pivot row	2	3	1	4	1	*0*	*0*	*0*	*0*	3
row4	1	0	2	2	0	*3*	*0*	*0*	*1*	9
optimize	-3	3	1	-2	0	*3*	*0*	*0*	*0*	2

problem with many zero values. Regarding this, it is a huge disadvantage that we can not update the tuplestore, because then we could just skip these columns and would only need to consider the columns, where the variable `valueInPivotRow` is not equal to zero. For both versions the costs would be much smaller if there are many zero values in the pivot row.
At this point we have updated the whole table once. Since we have new basic variables now, we need to update them at the end of every Simplex step. We just assign the pivot column as the new basic variable in the position of the pivot row in the basic variables array in our state node. Additionally we want to have the updated table in the initial tuplestore `tupstore1`. To do that, we copy the new table from the tuplestore `tupstore2` into the tuplestore `tupstore1` and free the allocated space in the tuplestore `tupstore2`. This also takes some mentionable time, but we can not describe it in a cost formula like we have done for the other operations. However, if the table is big enough, this time is marginal and hence we do not consider it in the cost calculations, but in small examples it probably has some influence on the total time.
After this we have done a whole Simplex step and the function can return the SimplexStatus `Running`, which means another Simplex step will be processed, because we have not found the optimum yet. This process will be repeated until no pivot column or no pivot row is found, which means we found the optimum or the problem is unbounded. In both cases we can return the results and the algorithm is finished. To return the values we can just assign the values at the right hand side to the current basic variable once and then return the correct values one by one by using the variable `numberOfReturnedTuples`.

5.3.4. Two phase method in PostgreSQL

As we have already seen many times some additional steps need to be processed for problems with no basic feasible solution at the beginning. We need to run the Simplex algorithm twice, first on an artificial problem to find a feasible solution and then on the result of the artificial problem with the real objective function. The extended version of the function `ExecSimplexMethod` is shown in Listing 5.16.

Listing 5.16: ExecSimplexMethod with Two Phase Method

```
TupleTableSlot *
ExecSimplexMethod(SimplexMethodState *node)
{
   if(node->currentStatus == RUNNING){
      // storing the input table in a tuplestore
      storeInTupleStore(node);

      if(node->startOfArtificials != -1){
         // preprocessing for the first phase
         initializePointers(node);
         transformObjectiveFunctionRow(node, false);

         // execute Simplex Method
         while(node->currentStatus == RUNNING){
            node->currentStatus = simplexStep(node, false);
         }

         if(node->currentStatus == OPTIMAL){
            // preprocessing for the second phase
            removeArtificialVariablesFromBasicVariables(node);
            transformObjectiveFunctionRow(node, true);
            node->currentStatus = RUNNING;
         }
         node->numberOfColumns = node->startOfArtificials;
      }

      // execute Simplex Method
      while(node->currentStatus == RUNNING){
         node->currentStatus = simplexStep(node, true);
      }
      getOutput(node);
   }
   // return the correct values
   return returnATuple(node);
}
```

Compared to the initial version in Section 5.3.3 we mainly added the code inside the if-clause, where the parameter `startOfArtificials` is compared with -1. This is the code, which belongs to the first phase and it is executed when an artificial variable was added during the method `storeInTupleStore`. In the explanation of the initial implementation in Section 5.3.3, we already mentioned that this method stores the scan relation on the state node into the tuplestore `node->tupstore1` and counts the number of rows and columns. Beside these tasks it also searches for the basic variables and adds an artificial variable for every row, where no basic variable could be found. Since in the initial implementation there was always a slack variable for every row, we moved the

explanation of this method in this Section. An overview of the tasks done in this method is shown in the following.

- Scan every tuple and put it on tuplestore
- Count number of rows and columns
- Store the name of the rows, if the sparse version is used
- Search for basic variables and store the rows they belong to
- Insert artificial variables for the rows, where no basic variable could be found

We scan every tuple in the input relation and put it on the tuplestore. While doing this, we also count the number of rows and columns in the relation and search for current basic variables. For every tuple we check if it belongs to a new column. When a new column appears `node->numberOfColumns` is increased by 1. For every tuple, which belongs to the first column (RHS), we increase `node->numberOfRows` by 1. For each other column than the right hand side we need to check if it is a basic variable. If the value in the objective function (first tuple of the column) is zero and only one non-zero value is contained in the other tuples of the column and this non-zero value is equal to 1, the current column is a basic variable. Hence we assign the position of this column in the array `node->basicVariables` as the basic variable for the row, where the value 1 belongs to.
After we have read every tuple in the relation, the scan part of the function is finished. In total we need to read and write every tuple once, which makes $m \cdot n$ tuples. Regarding the sparse version, this is done quite similar. The only difference is, that we additionally need to store the names of the rows. This can be done by reading the right hand side column once to count the number of rows, that we can allocate the needed space for the array `node->namesOfRows` and another time to store the names in the array. Hence in total we need to read the whole table once plus one additional time the right hand side column. This makes a total of $m \cdot n \cdot nz + m$ reads and $m \cdot n \cdot nz$ writes.
At this point the initial table is stored in the tuplestore and the position of the current basic variables are stored in the array `node->basicVariables`. If no basic variable was found for a row, the array contains the value 0 for this row. We now check for every row if the row has a basic variable already. If no basic variable is available, an artificial variable is added to serve as basic variable and the variable `node->numberOfColumns` is increased by one. If it is the first added artificial variable, the new column is marked as the start of the artificial columns in the variable `node->startOfArtificials`. If no artificial variable is added for any column, the variable `node->startOfArtificials` is assigned to -1. To insert the correct tuples in the tuplestore we insert a tuple with the value 0 for every row but the one the added artificial variable belongs to. In this row the value is 1. Since we have total control over the positioning of the tuples and the artificial tuples have no real meaning, the name of the row and column of these tuples does not matter and we just assign the value ”0” for the row and the column for all of

them. This is different in the sparse version. There we just need to insert two values, one for the row the artificial variable belongs to and one for the objective function row. Since we have the name of the rows stored in an array, we can use the correct names to store the tuples. To indicate that the column is an artificial column, the name of the row for the artificial objective function is "-123" for these columns.
If an artificial variable was added in this step, `node->startOfArtificials` is unequal to -1 and we need to apply the two-phase method and the function `transformObjectiveFunctionRow` is executed. Before that, we need to initialize the pointers, because we use them during the execution of this function. This function needs to create a new tuplestore as well, because some values are updated. However, most of the tuples are just copied from the old tuplestore on the new one. In the following an overview of the task is shown for this method.

- Get the values in the objective function for the basic variables
- Calculate the value of the relative cost coefficient row for every column
- Push the calculated value on the tuplestore along with the rest of the column

In the call of this function, we can see that besides of the `SimplexMethodState` a boolean is given as parameter to the function. This boolean stands for the current phase of the Simplex algorithm. If the boolean is `false` we are currently in the first phase and if it is `true` we are in the second phase. Regarding the function `transformObjectiveFunctionRow`, the only difference between the two phases is, that we need to consider the artificial objective function as row to transform if we are in the first phase, and otherwise the real objective function. Since we need to subtract every row multiplied by the value in the objective function of the corresponding basic variable, we need to get the values in the objective function for every basic variable at the beginning of this function. In the second phase we just restore every basic column and read the value in the first tuple, which represents the objective function. In the first phase, we assign the value 1 to every basic variable, which represents an artificial variable, and 0 for the other basic variables, since the artificial objective function yet does not exist and it is initialized like that. As the next step we can transform the objective function row into the relative cost coefficient row by subtracting every row from the objective function as many times as the value we previously stored for this row. We calculate the cost coefficient column by column and push the calculated value along with the other tuples of the corresponding column on the new tuplestore, to keep the ordering from the beginning. In the second phase we can delete the artificial cost coefficient row, which was created in the same method in the first phase. The artificial variables can be deleted out as well in the second phase by not pushing them on the tuplestore, because we do not need them again afterwards. After this function is called the first time we can optimize the artificial problem by calling the function `simplexStep`, which was introduced previously in Section 5.3.3. Again we need to signalize with a boolean if we are in the first or in the second phase, that we know if we need to optimize the artificial or the real problem. If the artificial problem is optimized and has an optimal solution not equal to zero, `not_feasible` is

returned as `SimplexState` and we can directly go to the end and return the result, that the problem is not feasible. Otherwise we have found a feasible solution for the real problem and can continue with the algorithm. The function `removeArtificialVariablesFromBasicVariables` checks if some artificial variables are still contained in the basic variables and exchange them by calling another Simplex step, with the row, where the artificial variable is the basic variable, as pivot row and a positive or negative value in a non-artificial column in this row as pivot column. Afterwards we can transform the real objective function by calling the function `transformObjectiveFunctionRow` again. The artificial variables are removed during that and the number of columns is reduced to the number of real variables. We can optimize the real problem and return the final results as done before.

6. Cost calculations and performance tests

Now that we have seen the implementations of the Simplex algorithm in PLSQL as well as in the PostgreSQL kernel with two different versions (sparse and dense), we want to compare them regarding efficiency and find their advantages and disadvantages.

6.1. Costs calculations of PostgreSQL implementations

Before we compare the running times of the different methods, we want to investigate the cost of the PostgreSQL implementations more deeply to finish the part of the optimizer. During the explanation of the implementation of a Simplex step, we described how many sequential read, random read, write, single get and single put operations each Simplex operation needs to execute in proportion to the number of rows and columns. At this point we want to sum up all these numbers in a single table to see how many sequential read, random read, write, single get and single put operations one iteration of the Simplex algorithm needs in total. Table 6.1 shows the different operations along with the amount of times they are called for the dense version and Table 6.2 the same for the sparse version. This table does not cover the cost for storing the table in the tuplestore at the beginning and computing the relative cost coefficients in the objective function row. Since these operations are just executed one or two times, while we have several iterations, these costs are marginal for most of the problems and we let them by the side for the following experiments and discussions. Beside this, the Tables 6.1 and 6.2 do not contain the cost to reset the tuplestore (clear `tupstore1` and assign $tupstore1 = tupstore2$) neither, because we could not calculate a cost formula for this operation. Since the time consumed by this operation is marginal regarding the total consumed time for bigger problems, this should not be a problem.
As mentioned in Section 5.3.3 the cost to update the normal columns is a bit higher in the sparse version than the cost presented in Table 6.2. This is because the parameter `nz` is higher in the part where we need to update the values (when the value in the pivot row is non-zero), than in the part where we can just read and put the tuples (when the value in the pivot row is zero). Through this we need to update a bit more tuples than we have calculated. Since these additional costs, which are added through that, are just minor and we only want to have an approximation of the total cost, we do not consider these costs in our cost formula to simplify our calculations. However, we keep in mind that the cost estimation for updating the columns is a bit optimistic and we discuss it in more details in Section 6.2 later.

Table 6.1.: Costs for one Iteration of the Simplex Algorithm in the Dense Version

operation	sequential read	random read	write
initializePointers	0	0	0
getPivotColumn	0	n	0
getPivotRow	0	$2m$	0
updatePivotColumn	m	0	m
updateNormalColumn	0	$2pr \cdot mn$	$pr \cdot mn$
Overall	0	$2pr \cdot mn$	$pr \cdot mn$

operation	sequential get	put
initializePointers	mn	0
getPivotColumn	0	0
getPivotRow	0	0
updatePivotColumn	0	0
updateNormalColumn	$(1.5 - pr)mn$	$(1 - pr)mn$
Overall	$(2.5 - pr)mn$	$pr \cdot mn$

To calculate the overall cost of one iteration, only the cost of the functions to initialize the pointers and to update the columns are considered, because they are the most expensive operations. In these two operations every tuple of the table needs to be processed, while in the other functions only the tuples of one or two rows respectively columns need to be investigated. If we have a big table these costs are getting marginal regarding the total costs.
At this point we only want to look at the representative total costs. We notice that the amount of random read, write and single put operations is always multiplied by `nz` in the sparse version compared to the dense version. Since `nz` is the percentage of non-zero values in the table, this value is between 0 and 1 and hence the sparse operation needs to process these operations less times. Considering the other two operations (sequential read and single get), we can see that the sparse version needs to sequentially read the same amount of tuples multiplied by `nz`, which the dense version only needs to get. Since the read operation consists of the get operation and of deserializing the tuple, the sparse version is worse in this case, if `nz` is high and deserializing a tuple has a considerable cost. This is because in the dense version the positioning of every tuple is exactly known, because every column has the same amount of tuples and when we want to skip some tuples we can do that without deserializing any tuples. In the sparse version we always need to deserialize a tuple, to know to which row and column it belongs. Summarized we see that the only operation, which the dense version executes less times than the sparse one, is the sequential read operation. However, to really compare the cost of the implementations, we need to compare the time cost of the different operations.
Before we consider all operations we want to investigate the random get operation, which is a part of the random read operation. The time consumed for a random get depends on the size of the table, hence we want to find out how the time for a random get changes when the table gets bigger. Figure 6.1 shows the consumed time regarding the size of

Table 6.2.: Costs for one Iteration of the Simplex Algorithm in the Sparse Version

operation	sequential read	random read	write
initializePointers	$mn \cdot nz$	0	0
getPivotColumn	0	n	0
getPivotRow	0	$2m \cdot nz$	0
updatePivotColumn	0	1	2
updateNormalColumn	$(1.5 - pr)mn \cdot nz$	$2pr \cdot mn \cdot nz$	$pr \cdot mn \cdot nz$
Overall	$(2.5 - pr)mn \cdot nz$	$2pr \cdot mn \cdot nz$	$pr \cdot mn \cdot nz$

operation	sequential get	put
initializePointers	0	0
getPivotColumn	0	0
getPivotRow	$(1 - nz)m$	0
updatePivotColumn	0	0
updateNormalColumn	0	$(1 - pr)mn \cdot nz$
Overall	0	$pr \cdot mn \cdot nz$

the table.
We can see that at first the time is constant until a size of around 200-300 tuples and then increasing logarithmically. This makes sense when we consider the buffer of the tuplestore, which can store around 200 tuples. Since a tuple from the current block in the buffer can be gotten without restoring the position of the file pointer in the disk, a tuplestore with less than 200 tuples never needs to restore the position of the file pointer, even when we have random gets. Therefore the time of a random get is almost constant there. If the tuple the active pointer points at is in the next block of 200 tuples in the disk, the tuplestore does not need to restore the position of the file pointer as well, it can just copy the next block into the buffer. Hence if we have a table with 400 tuples we have 2 full blocks of tuples (A and B) and 4 possible cases when we want to get the next tuple. The possible cases are shown in Table 6.3. If block A is currently stored in the buffer, the tuplestore never needs to restore the position of the file pointer, because it already has the tuples from block A in the buffer and block B is the next one on the disk. If block B is currently stored in the buffer, the tuplestore needs to restore the position of the file pointer if the tuple to get is in block A, because it needs to move the file pointer on the beginning again. If the tuple to get is in block B, it already has it in the buffer. So in one of four cases the tuplestore needs to restore the position of the file pointer on the position of the block, which contains the tuple the active pointer points at, and the probability is 25%. If we increase the size of the table, the probability that the tuplestore needs to restore the position increases as well. Because of this, the time needed for a random get operation increases logarithmically, at the beginning very fast and later it is almost constant again, since any way for almost every random get operation the position in the disk need to be restored.
Regarding the random get we will consider two cases. First if we have a small table with less than 200 tuples and second if we have a big table with more than 3000 tuples. In

Figure 6.1.: Time Consumed for a Random get Tuple Operation

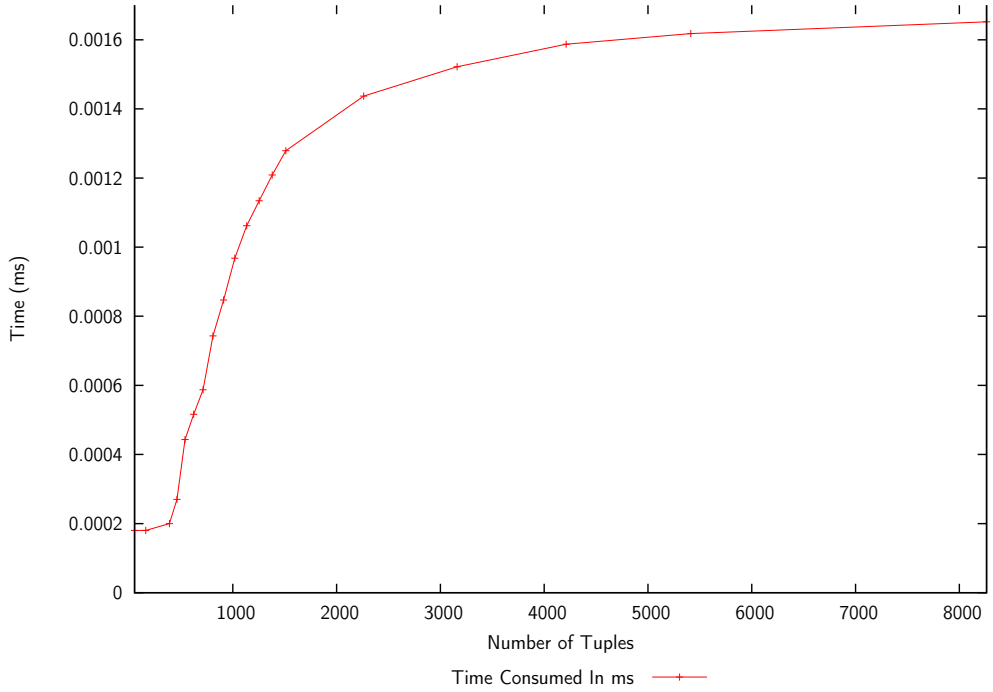


Table 6.3.: 4 Different Cases With 2 Blocks of Tuples

current block	needed block	restore position	probability
A	A	NO	25%
A	B	NO	25%
B	A	YES	25%
B	B	NO	25%

both cases the time consumed for a random get operation is almost constant. If we have a table between 200 and 3000 tuples we can look at the graph, how long one random get operation takes.
Now we can compare every operation and so compare the cost of the dense and the sparse implementations. We compare the average consumed time for every operation and multiply it with the amount of times the specific operations is executed. Since we only want to compare the two implementations and do not calculate exact times, we take the operation, which is the fastest one as reference and compare the other operations with it.
Table 6.4 shows the absolute time consumed for the elementary basic operations and the relative time compared to the sequential get operation, which is the fastest operation. In the following we will focus on the relative times.
Since in our implementations, we have combined some of the elementary basic operations to define new combined basic operations, we need to calculate the costs for these

Table 6.4.: Costs of Elementary Basic Operations

operation	absolute time consumed	relative time consumed
sequential get tuple	0.000180 ms	1 t_{sg}
random get tuple small relation	0.000180 ms	1 t_{sg}
random get tuple big relation	0.001740 ms	9.7 t_{sg}
put tuple	0.000215 ms	1.2 t_{sg}
deserialize	0.000942 ms	6.7 t_{sg}
serialize	0.001200 ms	5.2 t_{sg}

combined operations. Table 6.5 shows the costs for these combined operations along with the used elementary basic operations in relation to the cost to sequentially get a tuple.

Table 6.5.: Costs of Combined Basic Operations

operation	relative time consumed
sequential read	7.7 t_{sg}
random read small relation	7.7 t_{sg}
random read big relation	16.3 t_{sg}
write	6.4 t_{sg}
sequential get tuple	1 t_{sg}
put tuple	1.2 t_{sg}

With the values in Table 6.5, we are now able sum up the costs of the different operations according to Tables 6.1 and 6.2 and to calculate one single number for the cost of one iteration for the dense and the sparse implementation, which only relies on the parameters t_{sg}, `m`, `n`, `nz` and `pr`. We focus on big problems, hence we pick the cost for big relations(>3000 tuples) for the random read. In the dense version we have the following cost:

$$16.3 \cdot t_{sg} \cdot 2 \cdot pr \cdot mn + 6.4 \cdot t_{sg} \cdot pr \cdot mn + 1 \cdot t_{sg} \cdot (2.5 - pr) \cdot mn + 1.2 \cdot t_{sg} \cdot (1 - pr) \cdot mn$$

$$= ((36.8 \cdot pr + 3.7) \cdot t_{sg}) \cdot mn$$

In the sparse version we have the following cost:

$$7.7 \cdot t_{sg} \cdot (2.5 - pr) \cdot mn \cdot nz + 16.3 \cdot t_{sg} \cdot 2 \cdot pr \cdot mn \cdot nz +$$
$$6.4 \cdot t_{sg} \cdot pr \cdot mn \cdot nz + 1.2 \cdot t_{sg} \cdot (1 - pr) \cdot mn \cdot nz$$

$$= ((30.1 \cdot pr + 20.45) \cdot t_{sg}) \cdot mn \cdot nz$$

We can split the cost formulas into two parts. The left part, which is in the parenthesis, indicates the average cost of processing a tuple. It is dependent on the parameter `pr`. The remaining part on the right side indicates how many tuples are processed. It is dependent on the parameter `nz`. Both parameters have a value between 0 and 1. Regarding the

left part of the formulas, the dense version is strictly better than the sparse version, because the sparse version does not know the exact positioning of the tuples. If `pr` is near to zero, the dense version only need to process few operations to process a tuple, while the sparse version still needs to do a lot more operations. Hence the dense version is much better regarding the left part, if `pr` is low. However, the dense version can be in maximum around 5.5 times better than the sparse version when `pr` is near to zero. Regarding the right part of the formulas the sparse version is better than the dense one, because it has less tuples. In this part the difference between the two versions can differ infinitely, since `nz` can be close to zero. In total we observe that two parameters steer the performance difference between the two implementations, while `nz` is the dominant one. If `nz` is small enough, the sparse implementation is faster no matter what `pr` is. The threshold for `nz` is around 18%.
Since `pr` is the percentage of non-zero values in the pivot-row, it is dependent on the parameter `nz`. Hence it is most likely that if one of the parameter is close to zero also the other parameter is close to zero and vice versa. However, we already mentioned that we do not consider the parameters as the same, because they can differ extremely in some examples. In most cases we still have the tendency that if one of them is very high also the other is high and if one of them is very low also the other is low. Since we can not predict the exact values of these parameters before we have executed the Simplex algorithm, because they change during the execution, it makes it hard to predict which version is faster. To have the opportunity to still include these cost formulas in the optimizer we could consider `nz` and `pr` as the same and assign them to the initial percentage of non-zero values in the whole table, which is known from the beginning. We then are able to calculate the percentage of non-zero values at which the sparse version gets faster than the dense version, and pick the sparse implementation, if the initial `nz` is under this threshold. So we make an equality of the two cost formulas and consider $pr = nz$.

$$((36.8 \cdot nz + 3.7) \cdot t_{sg}) \cdot mn = ((30.1 \cdot nz + 20.45) \cdot t_{sg}) \cdot mn \cdot nz$$

$$\Rightarrow 36.8 \cdot nz + 3.7 = 30.1 \cdot nz^2 + 20.45 \cdot nz$$

$$\Rightarrow -30.1 \cdot nz^2 + 16.35 \cdot nz + 3.7 = 0$$

$$\Rightarrow nz = 0.717$$

Regarding this calculation, we see that the threshold of the dense and the sparse version is around 71.7% for `nz` and `pr`. This means that if `nz` is under 71.7% the sparse version should be faster, while if `nz` is over 71.7% the dense version should be faster, considering that $nz = pr$. This percentage seems quite high, because there are always basic variables with few non-zero values, and hence `nz` is probably smaller than this threshold. However, regarding the optimizer it possibly makes sense to pick the dense version, if the initial percentage of non-zero values is over this threshold and otherwise the sparse version.
In the next section we will consider different examples, calculate the percentage of non-zero and run both versions on them and compare the results and check if our cost calculations are correct.

6.2. Findings

Now that we know how the costs are composed for the C implementations, we want to run examples and count the total running times. We mainly focus on the C implementations, but we also want to compare them with the PLSQL implementation. We investigate 3 sort of problems. First we look at some benchmark problems from the Netlib.org library[1], second we investigate the transportation problem, where few non-zero values are present, and third a problem with an exponential amount of iterations, where the percentage of non-zero values in the pivot row(`pr`) is much lower that the percentage of non-zero values in the whole table(`nz`). In the last two sort of problems we concentrate on the C implementations with tuplestores, while in the benchmark problems, we compare the running times of all methods and also check the validity of the implementation.

6.2.1. Benchmark Problems

Correctness

Before we compare the running times, we want to check if our implementations are correct and deliver the correct results. To do so, we execute the Simplex algorithm on different Linear Programs using our implementations and the Python linear optimization library provided by SciPy, which applies the Simplex method as well. By doing this we can compare the results from the implementation in the SciPy library with the results from our implementation. If they are all the same our implementations should be correct. We run an example for every possible scenario, which can appear in the Simplex algorithm. We run the following problems:

1. a problem, which only requires to execute the second phase with a feasible solution
2. a problem, which requires to use the two-phase method with a feasible solution
3. a problem, which has no feasible solution
4. a problem, which has a positive unbound solution
5. a problem, which has a negative unbound solution
6. a problem, where artificial variables are contained in the basic variables after the first phase

Regarding the first problem, it needs to be mentioned, that the implementation in the SciPy library adds an artificial variable for every row, even when a basic variable is available. Because of this, the case, that the first phase is not executed, is not possible. However, since the result is the same at the end, we can still compare it with our implementations.
For every sort of problem listed above all implementations have the same result as the SciPy library showing they are correct, at least regarding small problem tables.

[1] http://www.netlib.org/lp/data/

Time Comparison

After we have proven the correctness of the implementations, we want to run some bigger examples with more iterations and compare the running times and the number of iterations. Table 6.6 shows different tested problems with its attributes while Table 6.7 shows the time consumed in total and Table 6.8 the time consumed per iteration and the number of needed iterations for these problems for the different implementations. Since the number of tuples is different for the first and the second phase, the average number of tuples during the execution is presented.

Table 6.6.: Attributes of Different Problems

problem	tuples	nz	pr
afiro	1607	0.1481	0.076
kb2	4729	0.2947	0.2746
share2b	16879	0.2507	0.1308
grow7	271454	0.1308	0.1592
agg	330413	0.0428	0.0252

Table 6.7.: Consumed Total Time (s) per Implementation

problem	SciPy	C array	C dense	C sparse	PLSQL
afiro	0.329	0.0065	0.051	0.036	0.476
kb2	0.430	0.018	1.389	1.184	36.5
share2b	0.69	0.064	6.524	5.654	209
grow7	11.6	1.5	146	90.5	no result
agg	15.1	1.1	42.6	13.0	2810

Table 6.8.: Consumed Time (ms) per Iteration / Number of Iterations

problem	SciPy	C array	C dense	C sparse	PLSQL
afiro	12.2 / 27	0.43 / 15	3.4 / 15	2.4 / 15	31.7 / 15
kb2	3.33 / 129	0.17 / 106	13.1 / 106	11.17 / 106	345 / 106
share2b	2.65 / 260	0.4 / 160	40.8 / 160	35.3 / 160	1304 / 160
grow7	14.9 / 776	4.66 / 316	462 / 316	287 /316	no result
agg	24.0 / 629	6.83 / 160	271.4 / 157	82.8 / 157	17563 / 160

In Table 6.7 we see that the PLSQL implementation is much slower than the other implementations in every tested example. It takes already quite long even for small examples and for big examples it takes so long, that it does not really makes sense to use this implementation and it possibly does not even work. For the problem `grow7` it took so long, that we have decided to stop the execution of this problem. The reason that the PLSQL implementation is so slow is because it needs to execute much more

operations on the table and therefore the table is accessed many times. In the other implementations we can simply scan through the table to get a new tuple, while in the PLSQL implementation we need to execute a whole query over the table. Hence there is no real reason to use the PLSQL implementation in practice.
Regarding the other implementations, we notice that the implementation using a C array and the implementation in the SciPy library are much faster for big examples than the implementations, which are using tuplestores to store the relation. The reason for this is, that they work in memory and never need to access the disk. Additionally, they do not always need to deserialize and serialize a tuple to get the information, because they do not work with binary values to store the tuples, they just store the real information in the memory. Hence the time needed to execute the Simplex algorithm with these implementations is much smaller. However, if the memory resources are limited, it is possible, that these implementations are using too much space for very big problem tables and they do not fit in the working memory any more. In such a case these implementations do not work. Considering such an example the C implementations using tuplestores should actually work, because the table is written to the disk and does not require much working memory space. Since with these implementations it would take very long to find the results for such big tables, it would not be that effective to use them as well. Interesting is that the implementation using the C array is even faster than the implementation in the SciPy library. One reason for that is, that the implementation in the SciPy library adds artificial variables for every row even when it is not needed. Because of this, it needs to execute more iterations than the other implementations to remove the artificial variables from the basic variables, which can be seen in Table 6.8. However, regarding the time needed per iteration the implementation using a C array in memory is still much faster than the implementation in the SciPy library, which is itself actually already really good. Another interesting point is, that in both implementations the time needed per iteration is decreasing at the beginning, even though that the size of the table is increasing. The reason for this is, that the preparation steps for the Simplex algorithm take an important part for these implementations and if we have just few iterations, which is the case at the beginning, the biggest part of the time per iteration comes from this. Considering the time needed for the dense and the sparse implementations using tuplestores, the sparse implementation is faster in every tested examples. Beside this, for both implementations we see that the time per iteration is smaller in the problem `agg` than in the problem `grow7` even thought that the problem `agg` has much more values. This is because the parameters `nz` and `pr` are very low in this example and regarding our cost formula the time consumed for a iteration is dependent on these parameters. These examples confirm this. We also see that `nz` and `pr` are always under 30%, so probably in bigger examples it is likely that these parameters are small and the sparse implementation is faster. To find that out, we will consider two sorts of examples and investigate only the running times of the dense and the sparse implementation.

6.2.2. Transportation Problem

The transportation problem is a problem that finds the minimum cost to transport an amount of goods from m origins to n destinations. Every origin contains an amount of a commodity, while every destination has a demand for the same commodity. More specifically, origin i can deliver a_i of the commodity and destination j needs b_j of it. We assume, that the total amount of goods in the origins is equal to the total amount of goods demanded by the destinations and hence

$$\sum_{i=1}^{m} a_i = \sum_{j=1}^{n} b_j.$$

Every transport from an origin i to a destination j has a cost c_{ij}. The goal is to find the cheapest transport pattern between origins and destinations, which satisfies all requirements. Regarding linear programs, we have the following objective function

$$\text{minimize} \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij}$$

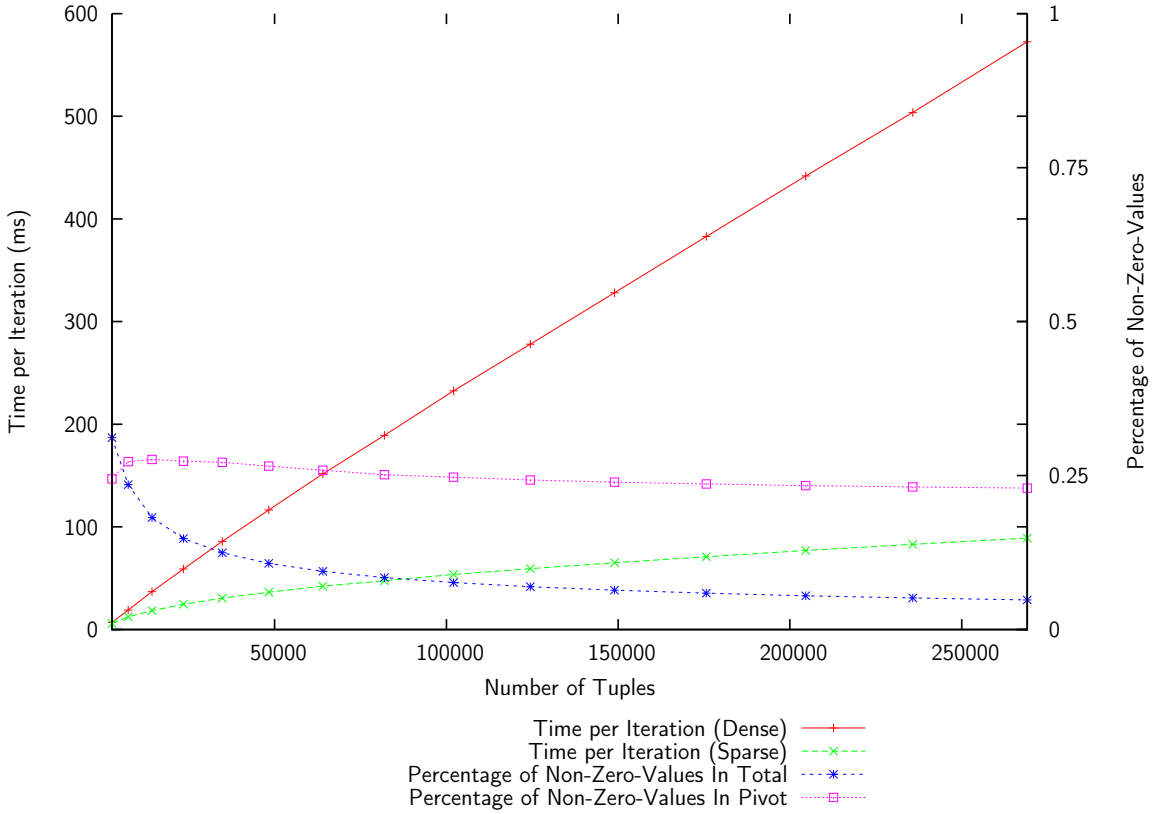
and for every origin and every destination one restriction. In total we have $m + n$ restrictions (rows) and $m \cdot n$ variables (columns). Every variable x_{ij} represents the amount of commodity that is transported from origin i to destination j. So, the column that corresponds to x_{ij} has non-zero values only in the row that corresponds to the constraint about origin a_i ($\sum_{j=0}^{n} x_{ij} = a_i$) and in the constraint about destination b_j ($\sum_{i=0}^{m} x_{ij} = b_j$). Along with the cost coefficient there are three non-zero values per column. So the initial sparsity of a transportation problem is computed as follows:

$$nz = \frac{3 \cdot m \cdot n}{(m+n) \cdot m \cdot n} = \frac{3}{m+n}$$

Since the transportation problem has only equality conditions, it needs an artificial variable for every constraint row. Artificial variables have only two non-zero values, namely the constraint row they belong to and the objective function (which is actually zero), hence the sparsity is still very low after adding them.
We assume that the percentage of non-zero values stays quite low for the transportation problem during the Simplex algorithm and hence we guess that the sparse version could be more efficient for this kind of problem, since there are a lot of zero values. We want to prove that by applying our algorithm on different transportation problems. We set the number of origins constantly equal to 10 and change the number of destinations during the experiments. We measure the time needed to execute the algorithm and divide it through the number of iterations needed. As result we get the time needed for one iteration of this problem. Like this we can better apply our cost calculations and compare the different experiments. Figure 6.2 shows the average time in milliseconds taken for one

iteration for the dense and the sparse implementations regarding the number of values the problem has, which is the product of the number of rows and columns. Besides this, the average percentage of `nz` and `pr` during the execution is shown for the different problems. The values of the percentages are shown in the right y-axis.

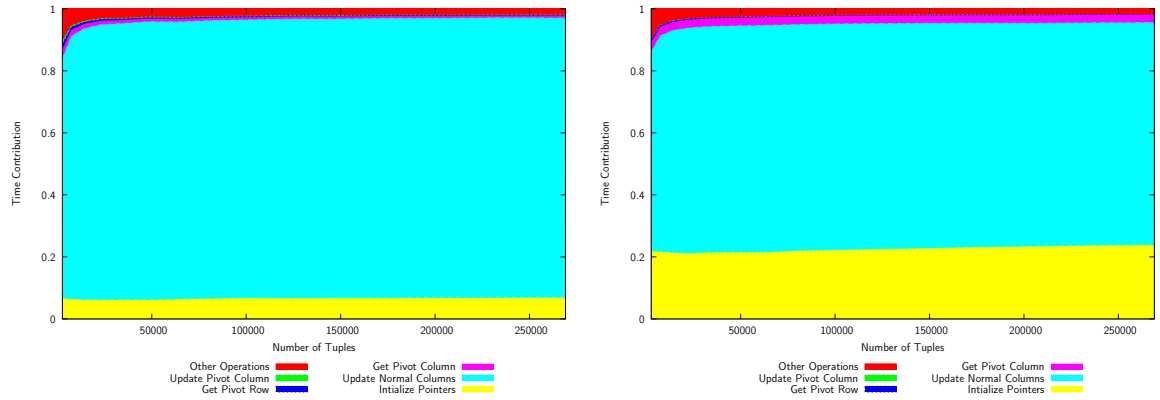
Figure 6.2.: Time per Iteration in the Transportation Problem



We see that the sparse implementation is faster than the dense one in every tested size of the problem, which makes sense, because the parameter `nz` is at maximum around 35% and nearly always lower than the parameter `pr`. `nz` is decreasing as bigger the table gets, which fits with the initial sparsity we calculated before. `pr` is decreasing as well, which makes sense, because it is related to `nz`. However, it is decreasing slower than `nz`. Therefore, regarding the cost calculations the sparse version should be much faster than the dense version, when the problem table is growing, because `nz` is very low for big tables. The experiments confirmed this: The proportion between the consumed times of the two implementations is increasing as bigger the problem table gets. While the consumed time of the dense version is increasing linearly regarding the size of the table, the grade of the time curve of the sparse version is decreasing slowly, because `nz` is decreasing.
We want to confirm these results of the consumed times by considering our cost formula.

To check if it is representative to only consider the function to initialize the pointers and to update the columns in the cost formula Figure 6.3 shows the contribution of the different Simplex operations of the total time for the dense and the sparse problem.
We see that the contribution of the total time of the functions to initialize the pointers

Figure 6.3.: Contribution of the Total Time of the Different Operations in the Transportation Problem



and to update the normal columns is near to 100% and it increases as bigger the table is. Therefore, our idea to only consider these operations to represent the total time is acceptable and it is more accurate as bigger the table is. The biggest contribution of the other part is the part called other operations, which represents the functions to store the relation in the tuplestore, to transform the objective function into the relative cost coefficients, to update the basic variables and to reset the tuplestores. The biggest part of it comes from the resetting of the tuplestores. We see, that even in such a big table, this operation has some influence on the total time, but it is not a problem to not consider it regarding the representative total cost.
At this point, we want to calculate the exact cost for an example, by inserting the parameters `nz` and `pr` in the cost formulas calculated before for both implementations. We pick the biggest problem tested, because we have seen that the bigger the table is the more exact the cost calculations should be. Beside the increasing contribution of the measured functions (initializing pointers and update other columns) also the cost for a random search operation should be more similar for the dense and the sparse implementation the bigger the table is. The investigated problem has $nz = 4.82\%$ and $pr = 22.96\%$. Inserted in the cost formulas, we get

$$(36.8 \cdot 0.2296 + 3.7) \cdot mn \cdot t_{sg} = 12.1493 \cdot mn \cdot t_{sg}$$

for the dense implementation and

$$= (30.1 \cdot 0.2296 + 20.45) \cdot mn \cdot 0.0482 \cdot t_{sg} = 1.3188 \cdot mn \cdot t_{sg}$$

for the sparse implementation. We can divide these costs through each other to get the proportion of them.

$$\frac{12.1493 \cdot mn \cdot t_{sg}}{1.3188 \cdot mn \cdot t_{sg}} = 9.212$$

Hence considering the cost formula the dense version should be around 9.212 times slower than the sparse version. By looking at the real times to solve the problem, the dense version took 572.63 ms and the sparse version 89.15 ms per iteration. We now can divide them through each other to get the real proportion of the two implementations.

$$\frac{572.63ms}{89.15ms} = 6.423$$

We see that the proportion of our cost formulas is around this number, hence the cost calculations we made are representative. The two proportions have a difference, because our cost formula is only a estimation for the total cost. We mentioned that the cost formula for the sparse problem is probably a bit optimistic, because the parameter `nz` is higher in the part, where we need to process more operations, but we did not consider that in the cost formula.
By inserting the number of tuples and the time needed for a sequential get in the costs above, we can even get an aggregated time for an iteration. The problem has 269082 tuples($m \cdot n$) and a sequential get takes 0.00018 ms. Inserted in the term above we get

$$12.1493 \cdot 269082 \cdot 0.00018ms = 588.448ms$$

for the dense implementation and

$$1.3188 \cdot 269082 \cdot 0.00018ms = 63.876ms$$

for the sparse implementation.
We see that the calculated time for the dense implementation is nearly the same as the real time, while the calculated time for the sparse implementation is a bit to low regarding the real time. This confirms our assumption, that the cost formula for the sparse problem is optimistic. Because of this the real proportion between the two versions is a bit lower than the calculated one, but since it is only an aggregation of the real cost this is acceptable.

6.2.3. Exponential Problem

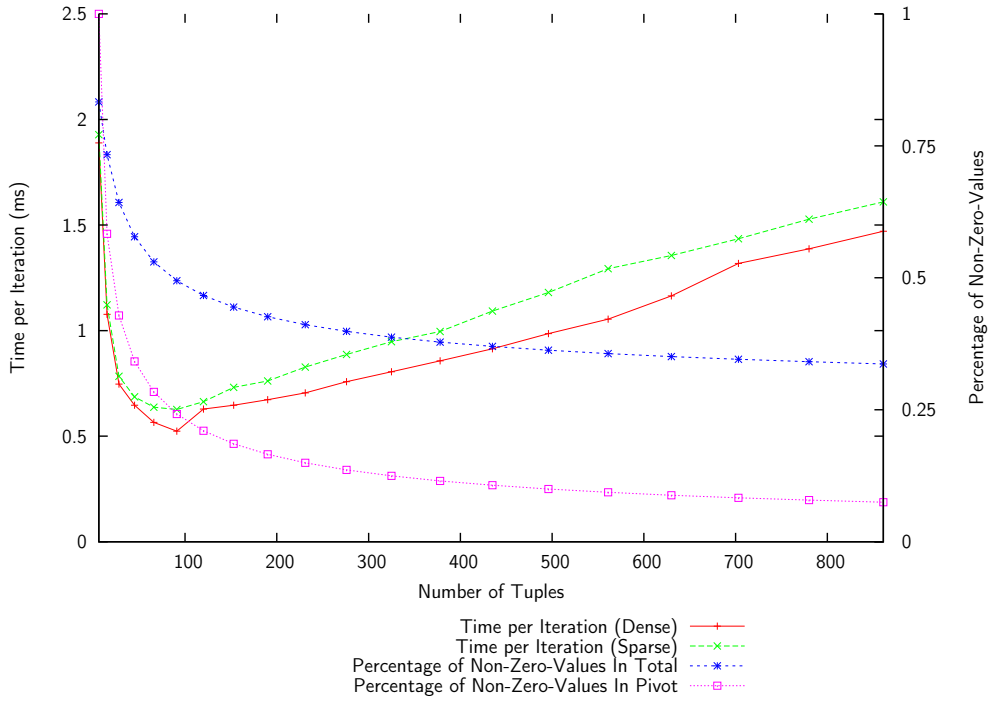
The worst case for the Simplex algorithm is if a problem has exponential number of iterations regarding the amount of variables and constraints. Victor Klee and George Minty [KM72] proved that by showing a class of linear problems, which requires an exponential number of iterations to solve an example of this class using the Simplex method. In this section we will look at such a problem with an exponential amount of iterations and execute some experiments. The characteristics of the problem type is shown in the following:

$$\text{maximize} \sum_{j=1}^{n} 10^{n-j} x_j$$

$$\text{subject to } 2 \sum_{j=1}^{i-1} 10^{i-j} x_j + x_i \leq 100^{i-1} \text{ for i = 1, ... , n}$$

To solve such a problem with the Simplex algorithm we need to perform $2^n - 1$ iterations. We will apply our implementations on such problems with different amounts of variables. We will execute problems with 1 until 20 variables, which means that in the last problem we need to perform $2^{20} - 1$ iterations. In this problem we have around 50% of non-zero values at the beginning, which is under 71.7%, so if we consider that `nz` and `pr` are the same, we should theoretically pick the sparse version. As done in the transportation problem, we measure the time needed to execute the algorithm and divide it through the number of iterations needed. Figure 6.4 shows the same graph as before in the transportation problem in Figure 6.2, but this time with the numbers from the exponential problem. The investigated problems are much smaller than in the transportation problem, because the amount of iterations grows exponentially and hence the times needed to execute a problem is always at least doubled when an additional variable is added. Since the problem tables are smaller also the times consumed for one iteration are smaller than in the transportation problem.

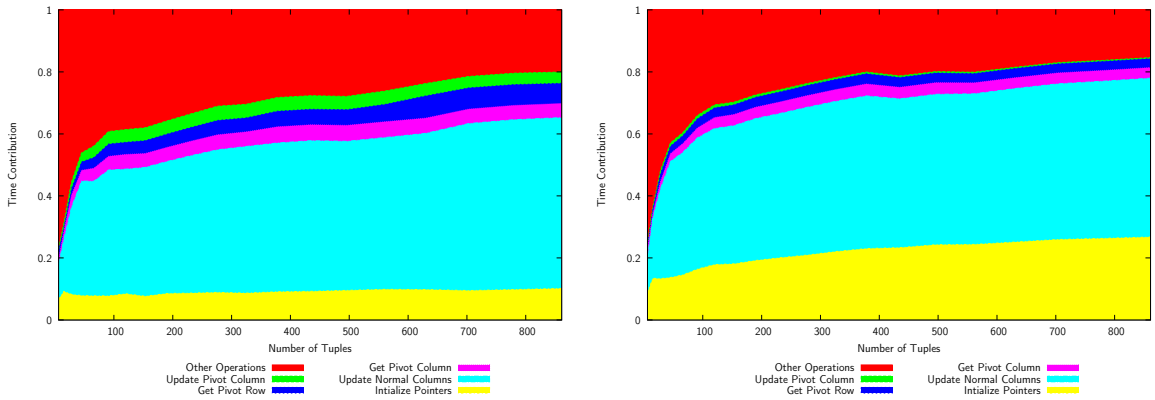
Figure 6.4.: Time per Iteration in the Exponential Problem



We see that the dense problem is faster than the sparse version, even though that the parameter `nz` is clearly under 71.7%. The reason for this is, that the parameter `pr` is much smaller than `nz`. While `pr` is around 10%, `nz` is around 35%, hence the advantage of the dense version regarding the average cost to process one tuple is bigger than the advantage of the sparse version that it operates on less tuples. However, a case that `pr` is so low, while `nz` is much higher is very rarely and never appeared in another tested

example. Regarding this, the difference between the consumed time of the dense and the sparse implementations is very small, they are even almost the same. Considering this, we assume that the sparse implementation is mostly better than the dense one and if it is not the dense implementation is just a bit faster.
As done in the transportation problem, we want to confirm the results of the consumed times by considering our cost formula. Since we have a much smaller table than in the transportation problem, we want to check if it is representative to only consider the function to initialize the pointers and to update the columns as the total time in this sort of problem as well. Figure 6.5 shows the contribution of the different Simplex operations of the total time for the dense and the sparse problem.

Figure 6.5.: Contribution of the Total Time of the Different Operations in the Exponential Problem



We see that the contribution of the total time of the functions to initialize the pointers and to update the normal columns is smaller in this problem than in the transportation problem. This is because the problem tables are much smaller here and hence the operations, which only process one or two rows/columns, and the resetting of the tuplestore in every iteration have much more influence on the total time. Mainly the resetting of the tuplestore took a lot of time and has high impact. However, because the time contribution of the two measured functions (initialize pointers and update columns) is increasing as bigger the examples are and their time contribution is similar in the two different implementations, the cost formulas can still be used to find out which implementation is faster and around how much faster it is. Therefore, our cost formulas are suitable to find out which implementation is faster, but not to calculate the exact total time they need to execute the algorithm, which is anyway not needed.
We want to prove this by looking at the cost formula for an individual example again. We again pick the biggest example, because the time contribution of the two measured functions is the highest. Since the table has less than 3000 tuples, the random read operation actually takes a bit shorter than in the cost calculations above. We will not consider this difference because it is quite small regarding the total time. For this example we

have $nz = 33.68\%$ and $pr = 7.5\%$. Inserted in the cost formulas, we get

$$(36.8 \cdot 0.075 + 3.7) \cdot mn \cdot t_{sg} = 6.46 \cdot mn \cdot t_{sg}$$

for the dense implementation and

$$= (30.1 \cdot 0.075 + 20.45) \cdot mn \cdot 0.3368 \cdot t_{sg} = 7.6479 \cdot mn \cdot t_{sg}$$

for the sparse implementation. And again we can divide these costs through each other to get the proportion of them.

$$\frac{6.46 \cdot mn \cdot t_{sg}}{7.6479 \cdot mn \cdot t_{sg}} = 0.8447$$

Hence considering the cost the dense version should take only around 84.47% of the time, which the dense version takes. Considering the real times, the dense version took 1.471 ms per iteration and the sparse version 1.610 ms per iteration. We divide them through each other to get the real proportion of the two implementations.

$$\frac{1.471ms}{1.610ms} = 0.9137$$

Again the proportion of our cost formula is near to the real proportion of the two implementations, hence the cost calculations we made are representative in this case as well. The main reason for the difference between the calculated proportion and the real one is that the cost formula only calculates an approximated cost and as we saw a big part of the total time is missing in this example.
By inserting the number of tuples in the costs above and the time needed for a sequential get as done in the transportation problem, we again can get an aggregated time for an iteration. The problem has 861 tuples($m \cdot n$) and a sequential get takes 0.00018 ms. Inserted in the term above we get

$$6.46 \cdot 861 \cdot 0.00018ms = 1.0012ms$$

for the dense implementation and

$$7.6479 \cdot 861 \cdot 0.00018ms = 1.185ms$$

for the sparse implementation.
We see that the calculated time for both implementations is smaller then the real time, while in the transportation problem the times were much more equal. This makes sense, regarding our findings from Figure 6.5. There we saw, that the function to initialize the pointers and to update the normal columns only contribute about 65% - 75% of the total time. If we add these missing parts to the calculated times, they are again really similar to the real time. This confirms our assumption, that the cost formulas are appropriated to find out which version is faster, but not to calculate the exact time in this example.

7. Conclusion

In the project we have implemented and discussed different approaches to integrate the Simplex algorithm in relational databases. In this section we want to summarize what we have found out and give a short conclusion. We implemented the Simplex algorithm using PLSQL and in the PostgreSQL kernel in the programming language C. Considering the C implementation, we had to solve the problem, that the intermediate results need to be materialised during the Simplex algorithm, because we need to read the tuples of each intermediate relation many times to apply the Simplex algorithm. To do so we have used two approaches. As a first approach we have stored the relation in a C array in memory and just updated the values in the array. However, we mainly focused on the second approach, where we stored the relation in a tuplestore, which is a generalized module for temporary tuple storage. In the second approach we implemented two different versions. In the first version every value of the problem table is stored in the tuplestore, while in the second version only the non-zero values are stored in the tuplestore. These versions are called dense and sparse version. For both versions we created a cost formula, which counts the number of basic operations need to be processed per iteration. We observed that in the dense version in average less operations need to be processed over each tuple of the relation than in the sparse version. This is because in the dense version the position of each tuple is exactly known in the relation, while in the sparse version we always have to deserialize a tuple to find out to which row and column it belongs to. However, in the sparse version the relation has less tuples, which have to be processed. Hence we could not directly say which version is the better one. We measured the average needed time for each basic operation, that we could sum up the different operations and find out which version is faster. We expected that the operations, which are operating with the disk are the most expensive one, but surprisingly deserializing and serializing a tuple is more expensive than get and put a tuple from the disk. Another interesting point was that the needed time for a random get operations increases logarithmically until around 10 times slower than a sequential get. After inserting the detected time for the operations, it was still hard to find out, which implementation is the better one, because other parameters affect the performance of the implementations, mainly the percentage of non-zero values in total and in the pivot rows. To compare the two implementations we have run different examples and measured the needed times. We also compared the times with the other 2 implementations and an additional implementation from the SciPy library. Before this, we have checked the correctness of the implementations and we could say that each implementation delivered correct results and could handle the different cases of the Simplex algorithm. Regarding the efficiency of the different implementations, the PLSQL implementation was much slower than the other implementations and it already took quite long for small examples and increased extremely when the examples were getting

bigger. Therefore we could say that the PLSQL implementation is not really useful and the worst of them. Regarding the two C implementations, which use a tuplestore to store the relation, the sparse version was mostly the faster one and if the dense version was faster the difference is just very small. Considering this, the sparse version should be preferred and probably be picked all the time, because the parameters, which are needed to calculate the cost can not be predicted before the implementation is executed and like this we are on the safe side, that we will not pick the much slower implementation. However, the implementation from the SciPy library and the C implementation using a C array in memory were much faster for the bigger examples than the implementations which are using a tuplestore, because they never need to access the disk and serialize and deserialize a tuple. Since the tuplestores are generally used to materialise a relation in PostgreSQL we expected the difference of the efficiency to be smaller and that the tuplestores work more efficient. However, the main problem is, that the tuplestore can not be updated, and we have to create a new tuplestore in every iteration. The implementation using the C array was even few times faster than the implementation from the SciPy, which is actually really good. Regarding this the best implementation for the Simplex algorithm is to use a C array in memory. This implementation can solve every tested problem in a very short time.

Bibliography

[KM72] Victor Klee and George J Minty. How good is the simplex algorithm. In *Inequalities III*, 1972.

[LY16] David G. Luenberger and Yinyu Ye. *Linear and Nonlinear Programming*, chapter Basic Properties of Linear Programs, pages 11–31. Springer International Publishing, 2016.

A. Contents of the CD-ROM

The CD-ROM contains the following content:

- **Zusfsg.txt** The abstract of this thesis in German.
- **Abstract.txt** The abstract of this thesis in English.
- **Bachelorarbeit.pdf** The digital copy of this thesis.
- **postgres** The code of the C implementations.
- **Simplex.SQL** The code of the PLSQL implementation.
- **diet_optimization_script.py** The script of the python implementation.
- **README** The instructions how to execute the implementations.
- **SQL_Scripts** Different SQL scripts, which are used to create optimization problems.
- **Experiments_Results** The results of the experiments in an Excel sheet and the in the report used data of it in csv format along with the created graphs.
- **Problems** The optimization problems used in the experiments in csv format.
- **report** The report in .tex format.