**Facharbeit im Nebenfach Informatik**

# Multiple linear regression in databases

**Sophie Leuenberger**

**08-926-362**

Supervised by Prof. Dr. M. Böhlen and O. Dolmatova
Department of Informatics, University of Zurich

23.07.2014

University of Zurich[UZH]

Department of Informatics

**Abstract**

Multidimensional statistical models such as multiple linear regression models are usually computed outside a data base management system (DBMS). In this report, we study how a multiple linear regression model can be computed inside a DBMS. The concept of *"summary matrices"*, which was introduced by Ordonez[5] as a tool to compute statistical models inside a DBMS, is presented and adapted for the case of multiple linear regression. We will consider two different approaches of implementation, study the performance of both of the alternatives, and figure out which one is better suited in which situation.

**Zusammenfassung**

Multidimensionale statistische Modelle werden gewöhnlich ausserhalb eines Datenbankmanagementsystems (DBMS) berechnet. In diesem Bericht untersuchen wir, wie man ein multiples lineares Regressionsmodell innerhalb eines DBMS berechnen kann. Das Konzept von *"summary matrices"*, das von Ordonez[5] als Werkzeug zur Berechnung von statistischen Modellen in Datenbanken eingeführt wurde, wird vorgestellt und für das Problem der linearen Regression angepasst. Wir werden zwei verschiedene Varianten der Implementierung betrachten, deren Leistung analysieren und erläutern, welche Variante in welcher Situation bessser geeignet ist.

# Contents

# List of Figures



# List of Tables

# 1 Introduction

Multidimensional statistical models describe relationships between variables by mathematical equations, which often can be reduced to linear algebra expressions over matrices. These models are usually computed outside a DBMS, by exporting the data to some statistical tool and then importing the results back to the DBMS. This is due to the fact that SQL, the standard language to process data inside a DBMS, has some limitations to perform complex matrix operations (see [5]). SQL does not naturally provide data types such as vectors or matrices, nor provide interfaces for linear algebra expressions (cf. [3]). The integration of statistical techniques into a DBMS is a very up-to-date subject, and will be studied in this report for a technique called *multiple linear regression*.

# 2 Preliminaries

## 2.1 Multiple linear regression

Multiple linear regression is a statistical technique that studies the relation between $d$ independent numeric variables $X_1, X_2, .., X_d$ and a single dependent variable $y$. The variables $X_1, X_2, .., X_d$ are usually called *predictor variables*, while the variable $y$ is called *response variable*.
To study the relation between the predictor variables and the response variable, data are measured and collected from $n$ observations. We will use the following notation:

| | |
|---|---|
| $x_{ik}$ | denotes the measured value of the $k$-th observation for predictor $X_i$ |
| $y_k$ | denotes the measured value of the $k$-th observation for the response $y$ |

The data set then forms the following $(d+1) \times n$ array:

| | Case Number | | | |
|---|---|---|---|---|
| | 1 | 2 | ... | $n$ |
| $X_1$ | $x_{11}$ | $x_{12}$ | ... | $x_{1n}$ |
| $X_2$ | $x_{21}$ | $x_{22}$ | ... | $x_{2n}$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| $X_d$ | $x_{d1}$ | $x_{d2}$ | ... | $x_{dn}$ |
| $y$ | $y_1$ | $y_2$ | ... | $y_n$ |

Table 1: Data set for multiple linear regression analysis

Given a data set as in Table 1, the multiple linear regression model assumes the relation between the response and the predictor variables to be linear, i.e. to satisfy the linear equations

$$y_i = \beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + ... + \beta_d x_{di} + e_i \quad (1)$$

for $i = 1, ..., n$ and unknown parameters $\beta_0, \beta_1, ..., \beta_d$. The term $\beta_0$ is called *intercept*, the unknown parameters $\beta_0, \beta_1, ..., \beta_d$ are called *regression coefficients* and the term $e_i$ measures the statistical error.

Matrix notation will simplify the computations used for regression analysis:
For

$$X := \begin{bmatrix} 1 & 1 & \dots & 1 \\ x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{d1} & x_{d2} & \dots & x_{dn} \end{bmatrix}, \quad \beta := \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_d \end{bmatrix}, \quad y := \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \quad \text{and} \quad e := \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_n \end{bmatrix}$$

the model equation becomes

$$y = X^T \beta + e.$$

In linear regression analysis, the number of observations $n$ is typically much bigger than the number of predictors $d$. Thus, the system of equations (1) is over-determined, i.e. has more equations than unknowns. Therefore, the regression coefficients need to be estimated from the empirical data. By Gauss-Markov theorem (cf. Weisberg[8], page 14), the best possible estimate for the regression coefficients is the so-called Ordinary Least Squares estimator

$$\hat{\beta} := (\hat{\beta}_0, \hat{\beta}_1, ..., \hat{\beta}_d)^T = (XX^T)^{-1} Xy,$$

providing that $(XX^T)^{-1}$ exists.

**Example 1.** *We consider a very small example with one single predictor variable and $n = 4$ observations. In practice, the sample sizes of course are much bigger.*
*We are interested in the relation between the body height and the weight of women. Let $X_1$ be the predictor variable measuring the body height and $y$ be the response variable measuring the weight.*

| | Case Number | | | |
|---|---|---|---|---|
| | 1 | 2 | 3 | 4 |
| $X_1$ | 170 | 154 | 164 | 158 |
| $y$ | 69 | 57 | 65 | 62 |

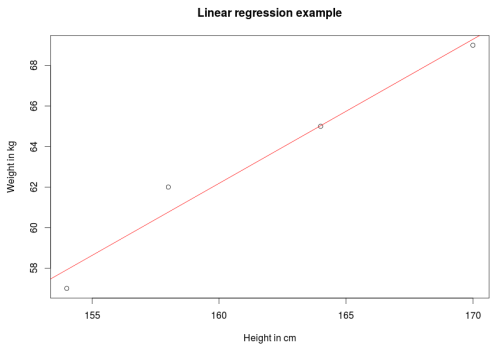


Figure 1: Linear regression example for $d = 1$ and $n = 4$

*In Figure 1, a data set table*[1] *and the corresponding scatter plot are shown. For* $d = 1$*, fitting a linear regression model means nothing else than laying a straight line through the data points. The red line represents the fitted regression line* $y = \hat{\beta}_0 + \hat{\beta}_1 x$*, where the regression coefficients are computed as shown below. The row of ones in the matrix* $X$ *serves for the integration of the intercept term* $\hat{\beta}_0$ *into the equation.*

$$(XX^T)^{-1}(Xy) = \left( \begin{bmatrix} 1 & 1 & 1 & 1 \\ 170 & 154 & 164 & 158 \end{bmatrix} \begin{bmatrix} 1 & 170 \\ 1 & 154 \\ 1 & 164 \\ 1 & 158 \end{bmatrix} \right)^{-1} \left( \begin{bmatrix} 1 & 1 & 1 & 1 \\ 170 & 154 & 164 & 158 \end{bmatrix} \begin{bmatrix} 69 \\ 57 \\ 65 \\ 62 \end{bmatrix} \right)$$

$$= \begin{bmatrix} 4 & 646 \\ 646 & 104476 \end{bmatrix}^{-1} \begin{bmatrix} 253 \\ 40964 \end{bmatrix} = \begin{bmatrix} -51.5578 \\ 0.7109 \end{bmatrix} = \begin{bmatrix} \hat{\beta}_0 \\ \hat{\beta}_1 \end{bmatrix} = \hat{\beta}$$

## 2.2 Terminology

Let us briefly summarize the terminology we will use in this report:

| | |
|---|---|
| $X$ | $(d+1) \times n$ matrix which contains the observed data for the predictors, plus an additional row of ones to integrate the intercept |
| $X_i$ | The $i$-th row vector of $X$, $X_i = (x_{i1}, x_{i2}, ..., x_{in})$ for $i = 1, ..., d$ |
| $y$ | The response variable |
| $\hat{\beta}$ | Vector of size $(d + 1)$, its coefficients are the best possible estimate for the unknowns in equation (1) |
| $Xy$ | Vector of size $(d+1)$, its coefficients will be denoted by $(Xy)_i$ for $i = 1, ..., (d+1)$ |
| $x_j$ | The $j$-th column vector of $X$, $x_j = (1, x_{1j}, x_{2j}, ..., x_{dj})^T$ for $j = 1, ..., n$<br>$x_j$ is also called *point*. |
| $\hat{X}$ | A $(d + 2) \times n$ matrix which contains the complete data set, i.e. a row of ones plus both the values for the predictors and for the response.<br>$\hat{X} := \begin{bmatrix} 1 & 1 & \dots & 1 \\ x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{d1} & x_{d2} & \dots & x_{dn} \\ y_1 & y_2 & \dots & y_n \end{bmatrix}$ |
| $\hat{X}\hat{X}^T$ | Matrix of size $(d + 2) \times (d + 2)$, the product of $\hat{X}$ with its transposed |

Table 2: Terminology

[1] The data set is totally fabricated.

# 3 Multiple linear regression in databases

## 3.1 Summary matrices

The main issue when building statistical models in databases is to perform complex matrix operations for matrices representing big data sets.

Ordonez[5] introduced two so-called *summary matrices* $L$ and $Q$, which compress the data given by $X$, but store all the information needed to build statistical models and score data. He defines the *linear sum of points*

$$L := \sum_{i=1}^{n} x_i$$

and the *quadratic sum of points*

$$Q := \sum_{i=1}^{n} x_i x_i^T$$

Assuming we have $d$ predictors, $L$ is a $(d+1)\times 1$, $Q$ is a $(d+1)\times(d+1)$ matrix. These matrices are much smaller than $X$ when $n \gg d$, but they summarize a lot of properties of $X$ which can be exploited by statistical techniques such as linear regression or factor analysis.

**Example 2.** *For* $X := \begin{bmatrix} 1 & 1 & 1 \\ 2 & 3 & 4 \end{bmatrix}$ *we have* $x_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$, $x_2 = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$ *and* $x_3 = \begin{bmatrix} 1 \\ 4 \end{bmatrix}$. *Then*

$$L = \begin{bmatrix} 1 \\ 2 \end{bmatrix} + \begin{bmatrix} 1 \\ 3 \end{bmatrix} + \begin{bmatrix} 1 \\ 4 \end{bmatrix} = \begin{bmatrix} 3 \\ 9 \end{bmatrix},$$

$$Q = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \begin{bmatrix} 1, 2 \end{bmatrix} + \begin{bmatrix} 1 \\ 3 \end{bmatrix} \begin{bmatrix} 1, 3 \end{bmatrix} + \begin{bmatrix} 1 \\ 4 \end{bmatrix} \begin{bmatrix} 1, 4 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix} + \begin{bmatrix} 1 & 3 \\ 3 & 9 \end{bmatrix} + \begin{bmatrix} 1 & 4 \\ 4 & 16 \end{bmatrix} = \begin{bmatrix} 3 & 9 \\ 9 & 29 \end{bmatrix}$$

We will adapt the idea of summary matrices, but alter it a bit since we are only interested in building linear regression models for this report. For this case, we can exploit the fact that the first row of $X$ only contains ones: As we can see in example 2, $L$ is already contained in $Q$. Therefore, a single summary matrix can be computed, containing both the linear sum of points and the quadratic sum of points. Furthermore, we will include the response vector $y$ into the summary matrix. Let $\hat{X}$ denote the matrix obtained from

$X$ by extending it with $y^T$, $\hat{X} := \begin{bmatrix} X \\ y^T \end{bmatrix}$ (see Table 2). Then we define

$$Q' := \hat{X}\hat{X}^T = \left[\begin{array}{cccc|c} n & \sum_{i=1}^{n} x_{1i} & \sum_{i=1}^{n} x_{2i} & \cdots & \sum_{i=1}^{n} x_{di} & (Xy)_1 \\ \sum_{i=1}^{n} x_{1i} & \sum_{i=1}^{n} x_{1i}^2 & \sum_{i=1}^{n} x_{1i}x_{2i} & \cdots & \sum_{i=1}^{n} x_{1i}x_{di} & (Xy)_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \sum_{i=1}^{n} x_{di} & \sum_{i=1}^{n} x_{di}x_{1i} & \sum_{i=1}^{n} x_{di}x_{2i} & \cdots & \sum_{i=1}^{n} x_{di}^2 & (Xy)_{d+1} \\ \hline (Xy)_1 & (Xy)_2 & & \cdots & & y^T y \end{array}\right]$$

(Q: $n$ to $\sum_{i=1}^{n} x_{di}^2$; Xy: $(Xy)_1$ to $(Xy)_{d+1}$)

Everything needed to construct the linear regression model from the summary matrix $Q'$ are the two submatrices $Q$ and $Xy$, which are highlighted above:

$$\hat{\beta} = Q^{-1}(Xy)$$

# 4 Implementation

In this section, two different approaches of computing the linear regression model are presented. The first approach is to implement a User-Defined Aggregate to compute the summary matrix, as Ordonez[5] suggests.
An aggregate function in SQL is a function that calls a transition function for each tuple of the table on which it is called. After processing every tuple, an optional final function can be called to compute the final results (see [6]).
The second approach is to implement a User-Defined Function. Both alternatives are written in PL/pgSQL.

First of all, let us define how matrices will be stored in the database.

## 4.1 Sparse matrix representation

A widely used way to represent matrices in databases is sparse matrix representation. A table corresponding to a matrix $M = (m_{ij})$ has three attributes, the first one storing the row number $i$, the second one storing the column number $j$ and the third one storing the value $m_{ij}$. A vector, which is a matrix with only one column, can also be stored in sparse representation, by omitting the attribute storing the column number.

For example, the matrix $M$ and the vector $v$

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \qquad v = \begin{bmatrix} 7 \\ 8 \\ 9 \end{bmatrix}$$

are stored as illustrated in figure 2.

```
 rownumber | colnumber | val
-----------+-----------+-----
         1 |         1 |   1
         1 |         2 |   2
         1 |         3 |   3
         2 |         1 |   4
         2 |         2 |   5
         2 |         3 |   6
(6 rows)
```

Representation of matrix $M$

```
 rownumber | val
-----------+-----
         1 |   7
         2 |   8
         3 |   9
(3 rows)
```

Representation of vector $v$

Figure 2: Sparse matrix representation

We will use this kind of matrix representation for this project. The advantages are that it is easy to process individual cells, and the number of both rows and columns can efficiently be increased without changing the whole structure of a table.

## 4.2 Implementation of a User-Defined Aggregate (UDA)

In this section, the first approach, implementing a User-Defined Aggregate, is presented. In a first step, all tables used in the algorithms are initialized, in particular the tables storing the values of the data set $\hat{X}$ and the table storing the values of the summary matrix $Q'$. These tables are created outside the linear regression aggregate, such that they are accessible within every algorithm that is part of the aggregate.

We then define a transition function that is called for every tuple in the table corresponding to the data set $\hat{X}$. Let us denote the access to a value in a table with square brackets, for example the value of $Q'$ with row number $i$ and column number $j$ will be denoted by $Q'[i][j]$ in the pseudo code.

**Algorithm 1** Transition function of the User-Defined Aggregate

**Input:** $i$: current row of $\hat{X}$, $j$: current column of $\hat{X}$, val: value at $\hat{X}[i][j]$
  **table** $temp(rowNumber, value) \leftarrow \quad (1, \hat{X}[1][j]),\ (2, \hat{X}[2][j]), ...,\ ((d+2), \hat{X}[d+2][j])$
  **for all** $k = 1, 2, ..., (d+2)$ **do**
    $Q'[i][k] \ = \ Q'[i][k] \ + \text{val} * \ temp[k]$
  **end for**

With this transition function, the entries of the summary matrix $Q'$ are updated incrementally. Assume the data set $\hat{X}$ is called `dataSet` and `(currentRow, currentCol, val)` denotes the tuple of $\hat{X}$ which is currently given as an input parameter to the transition function. Then the query in SQL to update $Q'$ would look like this:

**Listing 1** SQL query to update the summary matrix $Q'$

```
-- Compute entries of the summary matrix
FOR multVal, tempCol IN (SELECT dataSet.val, rowNumber FROM dataSet
                                WHERE colNumber = currentCol)
LOOP
  IF currentCol = 1 THEN
    INSERT INTO summaryMatrix VALUES ( currentRow, tempCol, val*multVal);
  ELSE
    UPDATE summaryMatrix SET s_val = s_val+(val*multVal)
    WHERE (s_row=currentRow AND s_col=tempCol);
  END IF;
END LOOP;
```

This query is quite time-consuming, since every time it gets called, the whole data set table has to be read to create a temporary table which stores the $d+2$ values we need to update $Q'$. We will analyze this problem further in section 4.2.1.

Once the transition function processed every row of $\hat{X}$, a final function is called to compute the regression coefficients vector $\hat{\beta}$. This computation consist of two steps: First, an LU-Decomposition (see Sauter[7]) is applied to the submatrix $Q$ of the summary matrix. We get

$$Q = LU$$

for a left lower triangular matrix $L$ and a right upper triangular matrix $U$.
In a second step, $\hat{\beta}$ can easily be computed from $L$ and $U$ solving the following system of equations:

$$\begin{cases} Lv = (Xy) \\ U\hat{\beta} = v \end{cases} \tag{2}$$

where $v$ is an auxiliary vector which is only used to find $\hat{\beta}$.
The algorithms for LU-Decomposition and for solving (2) are given in Algorithm 2 and Algorithm 3. The tables storing the values for $L$, $U$, $v$ and $\hat{\beta}$ are initialized before and are accessible in both of the algorithms.
The access to values in a table representing a vector will be denoted in the same manner as for matrices, but with only one square bracket. For example, the value of $v$ with row number $i$ will be denoted by $v[i]$.

**Algorithm 2** LU-Decomposition

**Input:** $d$: number of predictors
  **for all** $k = 1$ to $d + 1$ **do**
    $L[k][k] \leftarrow 1.0$
    $U[k][k] \leftarrow Q'[k][k]$
    **for all** $i = k + 1$ to $d + 1$ **do**
      $L[i][k] \leftarrow Q'[i][k] \div U[k][k]$
      $U[k][i] \leftarrow Q'[k][i]$
    **end for**
    **for all** $i = k + 1$ to $d + 1$ **do**
      **for all** $j = k + 1$ to $d + 1$ **do**
        $Q'[i][j] \leftarrow Q'[i][j] - L[i][k] * U[k][j]$
      **end for**
    **end for**
  **end for**

**Algorithm 3** Solve system of equations for $\hat{\beta}$

**Input:** $d$: number of predictors
  $v[1] = Q'[1][d + 2]$
  **for all** $k = 2$ to $d + 1$ **do**

$$v[k] \leftarrow Q'[k][d + 2] - \sum_{i=1}^{k-1} (v[i] * L[k][i])$$

  **end for**
  $\hat{\beta}[d + 1] \leftarrow v[d + 1] \div U[d + 1][d + 1]$
  **for all** $k = d$ to $1$ **do**

$$\hat{\beta}[k] \leftarrow \{v[k] - \sum_{i=k+1}^{d+1} (U[k][i] * \hat{\beta}[i])\} \div U[k][k]$$

  **end for**

### 4.2.1 Complexity analysis

Testing the algorithm on pseudo random created data sets (see section 5) demonstrated, that even for small data sets, the run time is very high. A possible reason could be that we store matrices in a completely different way than Ordonez[5] proposes.

Ordonez[5] suggests to store matrices not in sparse representation, but either using a string or a lists containing the elements of one column vector per tuple. The resulting data set table contains $n$ tuples, thus the transition function of a User-Defined Aggregate is called exactly $n$ times. In every step, the entries of the summary matrix can be updated using *only* values which are given as an input parameter to the function. In summary, for

the computation of the summary matrix, $n$ times $d^2$ products are computed and updated. Hence, time to compute the summary matrix is $O(nd^2)$.

In our case, using sparse matrix representation and including the response vector $y$ into the data set, the table representing the data set contains $n(d+2)$ tuples. Therefore, the transition function is called $n(d+2)$ times, where in every step, only one single value of the matrix is provided as an input parameter. Consequently, in every step, we first have to select $d+2$ additional values, which are needed to update $Q'$. For the selection of these $(d+2)$ values, we have to read the complete data set table, i.e. we have to read $n(d+2)$ values. Assuming the worst case scenario that for every read or write operation we have to access the disk, we can summarize the complexity analysis as following:

| | |
|---|---|
| Number of transition function calls | $n(d+2)$ |
| Number of I/O operations (Read)<br>(per step) | Read $n(d+2)$ values from the data set table to create temporary table<br>Read $(d+2)$ values from current summary matrix table |
| Number of arithmetic operations | (d+2) multiplications and (d+2) additions |
| Number of I/O operations (Write)<br>(per step) | Write $(d+2)$ values into temporary table<br>Write $(d+2)$ values to update summary matrix table |

We need $n(d+2)\{n(d+2)+(d+2)+(d+2)+(d+2)\}$ I/O operations, and the time to compute the summary matrix $Q'$ therefore is in $O(n^2d^2)$.
Time to compute the $LU$-Decomposition matrices $L$ and $U$ is $O(d^3)$, and to solve the system of equations (2) for $\hat{\beta}$ is $O(d^2)$.
Since typically $n \gg d$, the term $n^2d^2$ is dominating the term $d^3$. So we have a total runtime complexity of the User-Defined Aggregate lying in $O(n^2d^2)$.

This result is very bad, since we know from Ordonez[5] on the one hand, as from Markus Neumanns report[4] on the other hand, that it is possible to implement the computation of a linear regression model in $O(nd^2)$. Therefore, I would not suggest to use this code. The implementation of a User-Defined Aggregate is only reasonable if the values of the data set are present in form of column vectors, but not in form of sparse representation.

## 4.3 Implementation of a User-Defined Function (UDF)

In order to improve results, it might be better to abandon the User-Defined Aggregate, and instead implement a User-Defined Function which is better suited for matrices given

in sparse representation. We will rely on the following statement from Cohen et al.[1]:

> "[...] for example a sparse representation of the form (row number, column number, value). An advantage to this approach is that the SQL is much easier to construct for multiplication of matrices AB
>
> ```
> SELECT A.row_number, B.column_number, SUM(A.value * B.value)
> FROM A, B
> WHERE A.column_number = B.row_number
> GROUP BY A.row_number, B.column_number
> ```
>
> This query is very efficient on sparse matrices [...]"

According to this code, the summary matrix can be computed by one single, nested `SELECT` clause:

**Listing 2** Query to compute $Q'$

```
-- Compute entries of the summary matrix
INSERT INTO summaryMatrix
SELECT dataSet.rowNumber, transposed.rowNumber,
                          SUM(dataSet.val * transposed.val)
FROM dataSet, (SELECT colNumber, rowNumber, val FROM dataSet ) AS transposed
WHERE dataSet.colNumber = transposed.colNumber
GROUP BY dataSet.rowNumber, transposed.rowNumber;
```

The steps for computing $\hat{\beta}$ are implemented the same way as showed in section 4.2. A summary of the steps executed by the User-Defined Function is given by

**Algorithm 4** Linear regression function (UDF)

**Input:** $d$: number of predictors
**Compute** $Q' = \hat{X}\hat{X}^T$ according to Listing 2
$L, U \leftarrow$ LU_Decomposition($d$)
**Solve** $Lv = (Xy)$ for $v$
**Solve** $U\hat{\beta} = v$ for $\hat{\beta}$

### 4.3.1 Complexity analysis

Again, we assume the worst case scenario, that for every read or write operation, we have to access the disk. We need to read $n(d+2)$ values from the data set table to create a table storing the entries of $\hat{X}^T$:

| | |
|---|---|
| Number of I/O operations (Read) | Read $n(d+2)$ values from the data set table |
| Number of I/O operations (Write) | Write $n(d+2)$ values to create $\hat{X}^T$ -table |

Then, for every of the $(d+2)^2$ new tuples in the summary matrix table, we need to perform the following operations:

| | |
|---|---|
| Number of I/O operations (Read) | Read $2n$ values from the data set table and the $\hat{X}^T$ -table |
| Number of arithmetic operations | $n$ multiplications and $n-1$ additions |
| Number of I/O operations (Write) | Write one value to the summary matrix table |

In summary, we have $n(d+2)+n(d+2)+2n(d+1)^2$ I/O operations, so time to compute the summary matrix is in $O(nd^2)$. Time to compute the $LU$-Decomposition is $O(d^3)$, and for finding $\hat{\beta}$ is $O(d^2)$. Since $n \gg d$, the term $nd^2$ dominates the term $d^3$, hence the total run time complexity is $O(nd^2)$.

With the implementation using a User-Defined Function, we reach the same complexity as given by Ordonez[5]. Therefore, we can say that this implementation is a much more appropriate way to build the linear regression model when matrices are given in sparse representation.

# 5 Tests and Results

This section presents tests performed in PostgreSQL 9.3.4 running on a virtual Ubuntu 14.04 server. We used a virtual machine with two 2.4GHz dual core CPU's (QEMU Virtual CPU), 4 MB CPU cache and 8 GB RAM.

## 5.1 Tests on a random data set

The data sets used for testing were tables filled with pseudo random numbers in the interval $(0, 1]$, generated by the `Random()` function provided by PostgreSQL.

As mentioned in section 4.2, the implementation using User-Defined Aggregate leads to very poor results regarding the run time. Even for a small data set with a size of $n = 10'000$ and with number of predictors $d = 8$, the total time to compute the linear regression model was more than six hours.

The run time of the User-Defined Function turned out to be much smaller. Pseudo

random data sets were generated with sample sizes $n = 10^e$ for $e = 2, ..., 6$, and number of predictors $d = 8, 16, 32, 64$.
The average run times are given in Figure 3. The total time is denoted by "avg_totaltime", the time used for the $LU$-Decomposition and for finding $\hat{\beta}$ is denoted by "avg_lutime".

| r_samplesize | r_dimension | avg_lutime | avg_totaltime |
|---|---|---|---|
| 100 | 8 | 00:00:00.024769 | 00:00:00.034662 |
| 1000 | 8 | 00:00:00.023375 | 00:00:00.206945 |
| 10000 | 8 | 00:00:00.0244 | 00:00:02.587617 |
| 100000 | 8 | 00:00:00.022922 | 00:00:30.135406 |
| 1000000 | 8 | 00:00:00.047442 | 00:06:43.134087 |
| 100 | 16 | 00:00:00.306391 | 00:00:00.357453 |
| 1000 | 16 | 00:00:00.305108 | 00:00:01.014177 |
| 10000 | 16 | 00:00:00.279736 | 00:00:09.246475 |
| 100000 | 16 | 00:00:00.279783 | 00:01:52.660503 |
| 1000000 | 16 | 00:00:00.493563 | 00:43:55.148666 |
| 100 | 32 | 00:00:08.558576 | 00:00:08.752854 |
| 1000 | 32 | 00:00:08.174326 | 00:00:10.819981 |
| 10000 | 32 | 00:00:08.217107 | 00:00:40.413402 |
| 100000 | 32 | 00:00:08.629885 | 00:07:08.099398 |
| 1000000 | 32 | 00:00:11.138697 | 13:55:47.760526 |
| 100 | 64 | 00:06:04.531662 | 00:06:05.36333 |
| 1000 | 64 | 00:06:01.448873 | 00:06:11.951281 |
| 10000 | 64 | 00:06:11.347915 | 00:08:38.589029 |
| 100000 | 64 | 00:06:33.676965 | 01:26:50.43616 |

(18 rows)

Figure 3: Table of run times of the User-Defined Function for $d = 8, 16, 32, 64$

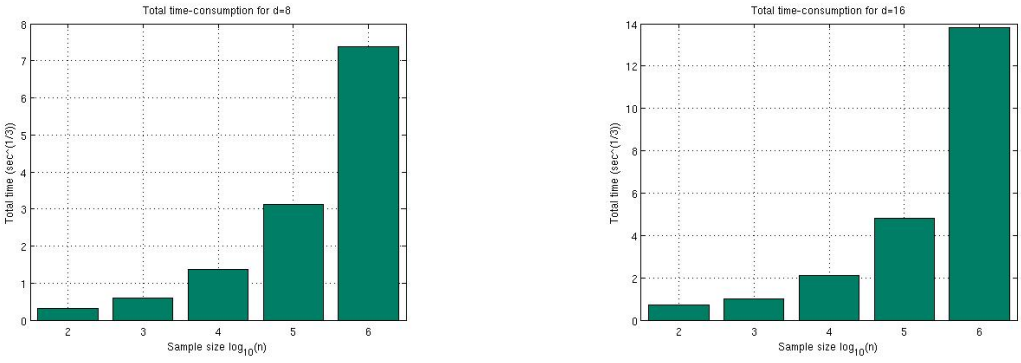


Figure 4: Run time plots of the User-Defined Function for $d = 8$ and $d = 16$, varying sample size $n$

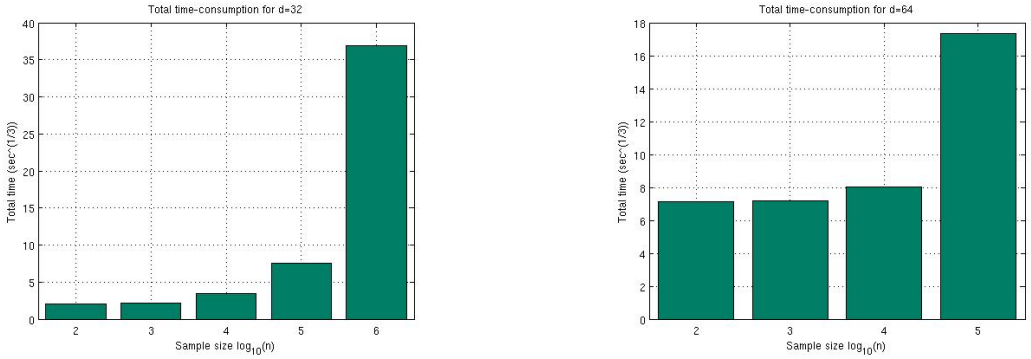


Figure 5: Run time plots of the User-Defined Function for $d = 32$ and $d = 64$, varying sample size $n$

The run time for both LU-Decomposition and solving the system of equations (2) for $\hat{\beta}$ only depends on the number of predictor variables $d$, it does not depend on the sample size $n$. This can be seen nicely in the following plot comparing the run time for these two parts of the algorithm for different values of $d$:

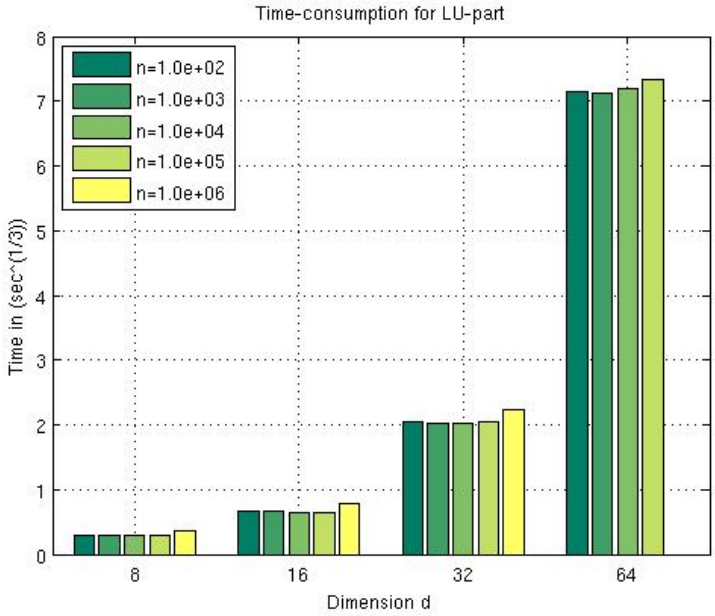


Figure 6: Time-consumption of the $LU$-part of the algorithm

Note that for $n = 1'000'000$ and $d = 64$, no tests were performed.

### 5.2 Tests on a constructed data set

To test both the correctness of the algorithm and the precision of the computer, we also tested the algorithm (UDF) for a constructed data set: Let

$$y_i = 1 + x_{1i} + x_{2i} + ... + x_{di}$$

for all $i \in \{1, 2, ..., n\}$, where $x_{ji}$ are randomly chosen for all $j \in \{1, 2, ..., d\}$. When we build the multiple linear regression model for this data set, we expect the coefficients of $\hat{\beta}$ all to be equal to 1.
Testing this for a sample size of $n = 100$ and number of predictors $d = 8$, the average of maximal deviation from 1 of the coefficients is in the area of $10^{-13}$, where the average is taken over ten runs. For $d = 16$, we have an average maximal deviation in the area of $10^{-12}$, for $d = 32$ near $10^{-11}$ and for $d = 64$ in the area of $10^{-10}$. These differences are probably due to the fact that $\beta_i$ is depending on $\beta_{i-1}$ for all $i \in \{1, 2, ..., d\}$. Hence, the more predictors we have, the more the deviation can grow.

```
 b1_samplesize | b1_dimension |     avg_maxerror
---------------+--------------+----------------------
           100 |            8 | 3.41981998275287e-13
           100 |           16 | 1.20776721956872e-12
           100 |           32 |  1.3533274501043e-11
           100 |           64 | 1.47207168676999e-10
(4 rows)
```

Figure 7: Table of average maximal deviations for $d = 8, 16, 32$ and 64, respectively

The deviations from 1 are very small, so we can be sure that the algorithms to build the multiple linear regression model are correct, and that the errors are due to the limited precision of the computer.

## 6 Conclusion and future work

Given a data set presented in sparse representation, we saw that implementing a User-Defined Aggregate does not make much sense, because the run time complexity is very high. The implementation of a User-Defined Function on the other hand turned out to be a reasonable way to compute a multiple linear regression model.
There are still possibilities to improve the algorithms. For example, one could exploit the fact that the summary matrix $Q'$ is symmetric. Based on this fact, only around half of the entries of the summary matrix really need to be computed.
A topic for future research could be the implementation of methods which, based on the computed model, score the data, for example simulate statistical tests or variable selection (for further information, see [8]).

# A R Code

The scatter plot shown in Example 1 was created in RStudio, Version 0.98.507.

```
X1 <- c(170, 154, 164, 158) # predictor
y <- c(69, 57, 65, 62) # response
plot(X1, y, main="Linear regression example", xlab="Height in cm",
                                                    ylab="Weight in kg")
abline(lm(y~X1), col="red") # regression line (weight~height)
lm(y~X1) # show OLS estimator
```

Listing 3: RstudioRegExa.R

# B SQL Code

## B.1 Initialization, data set creation and auxiliary functions

```
CREATE OR REPLACE FUNCTION CreateTables()
RETURNS void AS
$$
/*
*  Create all tables used in the algorithm.
*
*/
BEGIN
  DROP TABLE IF EXISTS dataSet;
  DROP TABLE IF EXISTS summaryMatrix;
  DROP TABLE IF EXISTS LU_Dec_L;
  DROP TABLE IF EXISTS LU_Dec_U;
  DROP TABLE IF EXISTS v_vec;
  DROP TABLE IF EXISTS beta;

  CREATE TABLE dataSet (rowNumber INTEGER, colNumber INTEGER, val DOUBLE PRECISION);
  CREATE TABLE summaryMatrix (s_row INTEGER, s_col INTEGER, s_val DOUBLE PRECISION);
  CREATE TABLE LU_Dec_L (l_row INTEGER, l_col INTEGER, l_val DOUBLE PRECISION);
  CREATE TABLE LU_Dec_U (u_row INTEGER, u_col INTEGER, u_val DOUBLE PRECISION);
  CREATE TABLE v_vec (v_vec_row INTEGER, v_vec_val DOUBLE PRECISION);
  CREATE TABLE beta(beta_row INTEGER, beta_val DOUBLE PRECISION);
END;
$$ LANGUAGE plpgsql;
```

Listing 4: createTables.sql

```
CREATE OR REPLACE FUNCTION createRandomTable(sampleSize INTEGER, dimension INTEGER,
  distribution INTEGER DEFAULT 1, minValue DOUBLE PRECISION DEFAULT 0.0,
  maxValue DOUBLE PRECISION DEFAULT 1.0)
RETURNS void AS $$
/*
*  Create a table representing a data set
*  Distribution:        1   Uniform
*                       2  Constructed (beta=1)
*/
DECLARE
  maxMinusMin DOUBLE PRECISION;
  maxMinusMinI INTEGER;
  tempY DOUBLE PRECISION;
BEGIN
  DELETE FROM dataSet;

  IF distribution = 1 THEN
  -- uniform distribution
    maxMinusMin := maxValue-minValue;
    FOR colNumber in 1..sampleSize
    LOOP
      -- Insert values belonging to the intercept beta_0:
      INSERT INTO dataSet VALUES (1, colNumber, 1.0);

      FOR rowNumber in 2..dimension+1
      LOOP
        --Insert values belonging to the regressors x1, x2, ... , xd:
        INSERT INTO dataSet
        VALUES (rowNumber, colNumber, minValue + maxMinusMin * Random());
      END LOOP;
      -- Insert values belonging to y:
      INSERT INTO dataSet
      VALUES (dimension+2, colNumber, minValue + maxMinusMin * Random());
    END LOOP;

  ELSIF distribution = 2 THEN
  -- Constructed data set (beta=1)
    maxMinusMinI := floor(maxValue)-ceil(minValue);
    FOR rNum in 2..dimension+1
    LOOP
      FOR cNum in 1..sampleSize
      LOOP
        INSERT INTO dataSet
```

```
        --Insert values belonging to the regressors x1, x2, ... , xd:
        VALUES (rNum, cNum, ceil(minValue + maxMinusMinI * Random()));
      END LOOP;
    END LOOP;
    FOR cNum in 1..sampleSize
    LOOP
      -- Insert values belonging to the intercept beta_0:
      INSERT INTO dataSet VALUES (1, cNum, 1);
      SELECT SUM(dataSet.val) INTO tempY
      FROM dataSet where dataSet.colNumber = cNum;
      -- Insert values belonging to y:
      INSERT INTO dataSet VALUES (dimension+2, cNum, tempY);
    END LOOP;
  END IF;
END;
$$ LANGUAGE plpgsql;
```

Listing 5: CreateRandomDataSet.sql

```
CREATE OR REPLACE FUNCTION saveCSV(sampleSize INTEGER, dimension INTEGER,
        distribution INTEGER, counter INTEGER DEFAULT 1 )
RETURNS void AS $$
/*
*  Save the tested data set in a CVS
*  Distribution:  1  Uniform, 2 Constructed (beta=1)
*/
DECLARE
  qry TEXT;
  filename TEXT;
BEGIN
  filename := format('/tmp/test_%s_%s_%s_%s_dataSet.csv', sampleSize,
  dimension, distribution, counter);
  qry := FORMAT('COPY (select * from dataSet ORDER BY rowNumber, colNumber)
                 TO %L CSV HEADER', filename);
  EXECUTE qry;
  filename := format('/tmp/test_%s_%s_%s_%s_beta.csv', sampleSize,
  dimension, distribution, counter);
  qry := FORMAT('COPY (select * from beta ORDER BY beta_row)
                 TO %L CSV HEADER', filename);
  EXECUTE qry;
END;
$$ LANGUAGE plpgsql;
```

Listing 6: saveCSV.sql

```
CREATE OR REPLACE FUNCTION LUDecomposition(dimension INTEGER)
RETURNS void AS $$
/*
*  LU Decomposition factors matrix Q into a left lower triangular matrix L
*  and a right upper triangular matrix U. LU Decomposition matrices are used to
*  solve the linear system of equations (XX^t) * beta = (XY^t) for beta.
*/
DECLARE
  -- Declare variables to store temporary results:
  tempS DOUBLE PRECISION;
  tempL DOUBLE PRECISION;
  tempU DOUBLE PRECISION;
  tempBeta DOUBLE PRECISION;
  tempSum DOUBLE PRECISION;
  tempV DOUBLE PRECISION;
  tempRow INTEGER;
  tempCol INTEGER;
  tempVal DOUBLE PRECISION;
BEGIN
  -- Compute entries of L and U:
  FOR k IN 1..dimension+1
  LOOP
    INSERT INTO LU_Dec_L VALUES (k,k,1.0);
    SELECT s_val INTO tempS FROM summaryMatrix WHERE s_row = k AND s_col = k;
    INSERT INTO LU_Dec_U VALUES(k,k, tempS);

    SELECT u_val INTO tempU FROM LU_Dec_U WHERE u_row = k AND u_col = k;
    FOR i IN k+1..dimension+1
    LOOP
      SELECT s_val INTO tempS FROM summaryMatrix WHERE s_row = i AND s_col = k;
      INSERT INTO LU_Dec_L VALUES (i, k, tempS / tempU);
      SELECT s_val INTO tempS FROM summaryMatrix WHERE s_row = k AND s_col = i;
      INSERT INTO LU_Dec_U VALUES (k, i, tempS);
    END LOOP;
    FOR i IN k+1..dimension+1
    LOOP
      SELECT l_val INTO tempL FROM LU_Dec_L WHERE l_row=i AND l_col = k;
      FOR j IN k+1..dimension+1
      LOOP
        SELECT s_val INTO tempS FROM summaryMatrix
        WHERE s_row = i AND s_col = j;
        SELECT u_val INTO tempU FROM LU_Dec_U
```

```
      WHERE u_row = k AND u_col = j ;
      UPDATE summaryMatrix SET s_val = s_val-(tempU * tempL)
      WHERE s_row = i AND s_col = j;
    END LOOP;
  END LOOP;
END LOOP;

-- Solve system of linear equations to get beta vector:
SELECT s_val INTO tempS FROM summaryMatrix
WHERE s_row = 1 AND s_col = dimension+2;
INSERT INTO v_vec VALUES(1, tempS);
-- Compute v_vec where L * v_vec = XY^t:
FOR k IN 2..dimension+1
LOOP
  tempSum:=0;
  FOR i IN 1..k-1
  LOOP
    SELECT l_val INTO tempL FROM LU_Dec_L WHERE l_row=k AND l_col = i;
    SELECT   v_vec_val INTO tempV FROM v_vec WHERE v_vec_row = i;
    tempSum := tempSum + (tempV * tempL);
  END LOOP;
  SELECT s_val INTO tempS FROM summaryMatrix
  WHERE s_row = k AND s_col = dimension+2;
  INSERT INTO v_vec VALUES(k, tempS - tempSum);
END LOOP;

-- Compute beta where U * beta = v_vec:
SELECT v_vec_val INTO tempV FROM v_vec WHERE v_vec_row=dimension+1;
SELECT u_val INTO tempU FROM LU_Dec_U
WHERE u_row=dimension+1 AND u_col=dimension+1;
INSERT INTO beta VALUES(dimension+1, tempV / tempU);
FOR k IN REVERSE dimension..1
LOOP
  tempSum:=0;
  FOR i IN k+1..dimension+1
  LOOP
    SELECT u_val INTO tempU FROM LU_Dec_U WHERE u_row = k AND u_col = i;
    SELECT   beta_val INTO tempBeta FROM beta WHERE beta_row = i;
    tempSum := tempSum + tempBeta * tempU;
  END LOOP;

  SELECT u_val INTO tempU FROM LU_Dec_U WHERE u_row = k AND u_col = k;
  SELECT v_vec_val INTO tempV FROM v_vec WHERE v_vec_row=k;
```

```
    INSERT INTO beta VALUES (k, (tempV - tempSum)/ tempU);
  END LOOP;
END;
$$ LANGUAGE plpgsql;
```

Listing 7: LU_Decomposition.sql

## B.2 SQL Codes for the UDA

```
CREATE AGGREGATE LinReg(INTEGER, INTEGER, DOUBLE PRECISION)(
  STYPE = INTEGER,
  SFUNC = LinRegStep,
  FINALFUNC = LinRegFinal
);
```

Listing 8: aggragate.sql

```
CREATE OR REPLACE FUNCTION LinRegStep(aggregateState INTEGER, currentRow INTEGER,
                                      currentCol INTEGER, val DOUBLE PRECISION)
RETURNS INTEGER AS $$
/*
*  Stepfunction of the aggregate function.
*  Entries of the summary matrix Q' are incrementally updated.
*/
DECLARE
  multVal DOUBLE PRECISION;
  tempCol INTEGER;
BEGIN
  -- Compute entries of the summary matrix
  FOR multVal, tempCol IN
  (SELECT dataSet.val, rowNumber FROM dataSet WHERE colNumber = currentCol)
  LOOP
    IF currentCol = 1 THEN
      INSERT INTO summaryMatrix VALUES ( currentRow, tempCol, val*multVal);
    ELSE
      UPDATE summaryMatrix SET s_val = s_val+(val*multVal)
      WHERE (s_row=currentRow AND s_col=tempCol);
    END IF;
  END LOOP;
  RETURN currentRow;
END;
$$ LANGUAGE plpgsql;
```

Listing 9: stepfunction.sql

```
CREATE OR REPLACE FUNCTION LinRegFinal(dimension INTEGER)
RETURNS INTERVAL AS $$
/*
*  The final function of the aggragate function.
*  The entries of the summary matrix are used to find a solution beta
*  for the multiple linear regression equation y = beta^t * X.
*/
DECLARE
  time1 TIMESTAMP;
  time2 TIMESTAMP;
  LU_executionTime INTERVAL;
BEGIN
  time1 := clock_timestamp();
  PERFORM LU_Decomposition(dimension-2);
  time2 := clock_timestamp();
  LU_executionTime = time2-time1;
  RETURN LU_executionTime;
END;
$$ LANGUAGE plpgsql;
```

Listing 10: finalfunction.sql

```
CREATE OR REPLACE FUNCTION testFunc(sampleSize INTEGER, dimension INTEGER,
                distribution INTEGER, minValue DOUBLE PRECISION DEFAULT 0.0,
                maxValue DOUBLE PRECISION DEFAULT 1.0)
RETURNS void AS $$
/*
*  Function to test the aggregate function LinReg(INTEGER, INTEGER, DOUBLE PRECISION).
*  Distribution:  1  Uniform
*                       2  Constructed (beta=1)
*  minValue, maxValue: Define interval for random numbers
*/
DECLARE
  tempCounter INTEGER;
  startTime TIMESTAMP;
  endTime TIMESTAMP;
  LU_executionTime INTERVAL;
  total_executionTime INTERVAL;
  filename TEXT;
  qry TEXT;
```

```
  tempInt INTEGER;
BEGIN
  PERFORM CreateTables();
  PERFORM createRandomTable(sampleSize, dimension, minValue, maxValue);
  startTime = clock_timestamp();
  SELECT linReg(rowNumber, colNumber, val) INTO LU_executionTime from dataSet;
  endTime = clock_timestamp();
  total_executionTime := endTime - startTime;
  SELECT count(*) INTO tempCounter FROM resultTable
  WHERE R_sampleSize = sampleSize AND R_dimension = dimension
  AND R_distribution = distribution;
  PERFORM saveCSV(sampleSize, dimension, distribution, tempCounter+1);

  INSERT INTO resultTable
  VALUES (sampleSize, dimension, distribution, tempCounter+1,
                total_executionTime, LU_executionTime);

  filename := format('/tmp/test_%s_%s_%s_%s_tableSizes.csv', sampleSize,
                dimension, distribution, tempCounter+1);
  qry := FORMAT('COPY (SELECT * FROM
  (SELECT
  relname as "Table",
  pg_size_pretty(pg_total_relation_size(relid)) As "Size",
  pg_size_pretty(pg_total_relation_size(relid) - pg_relation_size(relid))
  as "External Size"
  FROM pg_catalog.pg_statio_user_tables ORDER BY pg_total_relation_size(relid) DESC)
  AS tempInt ) TO %L CSV HEADER', filename);

  EXECUTE qry;
END;
$$ LANGUAGE plpgsql;
```

Listing 11: testFunc.sql

## B.3 SQL Codes for the UDF

```
CREATE OR REPLACE FUNCTION linReg(dimension INTEGER)
RETURNS INTERVAL AS $$
/*
*  Compute the linear regression coefficients beta={beta_0, beta_1,..., beta_d}
*  for a multiple linear regression model y = beta * X
*/
DECLARE
```

```
  LUtime_start TIMESTAMP;
  LUtime_end TIMESTAMP;
  LUtime INTERVAL;
BEGIN
  -- Compute entries of the summary matrix
  INSERT INTO summaryMatrix
  SELECT dataSet.rowNumber, transposed.rowNumber,
         SUM(dataSet.val * transposed.val)
  FROM dataSet, (SELECT colNumber, rowNumber, val FROM dataSet) AS transposed
  WHERE dataSet.colNumber = transposed.colNumber
  GROUP BY dataSet.rowNumber, transposed.rowNumber;

  -- Compute the linear regression coefficients vector
  -- beta = {beta_0, beta_1,..., beta_d} using LU_decomposition
  LUtime_start := clock_timestamp();
  PERFORM LUDecomposition(dimension);
  LUtime_end := clock_timestamp();
  LUtime = LUtime_end-LUtime_start;

  RETURN LUtime;
END;
$$ LANGUAGE plpgsql;
```

Listing 12: LinReg.sql

```
CREATE OR REPLACE FUNCTION testFunc(sampleSize INTEGER, dimension INTEGER,
        distribution INTEGER DEFAULT 1, minValue DOUBLE PRECISION DEFAULT 0.0,
        maxValue DOUBLE PRECISION DEFAULT 1.0)
RETURNS void AS $$
/*
*  Function to test the linear regression function
*  Distribution:  1  Uniform
*      2  Constructed (beta=1)
*  minValue, maxValue: Define interval for random numbers
*/
DECLARE
  tempCounter INTEGER;
  linReg_start TIMESTAMP;
  linReg_end TIMESTAMP;
  LU_time INTERVAL;
  total_time INTERVAL;
  filename TEXT;
  qry TEXT;
```

```
  tempAlias INTEGER;
BEGIN
  -- Initialization:
  PERFORM CreateTables();
  PERFORM createRandomTable(sampleSize, dimension,
                                distribution, minValue, maxValue);

  -- Find linear regression coefficients:
  linReg_start = clock_timestamp();
  select linReg(dimension) INTO LU_time from dataSet;
  linReg_end = clock_timestamp();
  total_time := linReg_end - linReg_start;

  -- Save dataSet and result vector to a file:
  SELECT count(*) INTO tempCounter FROM resultTable
  WHERE R_sampleSize = sampleSize AND R_dimension = dimension
                                AND R_distribution = distribution;
  PERFORM saveCSV(sampleSize, dimension, distribution, tempCounter+1);

  -- Store performance information:
  INSERT INTO resultTable VALUES (sampleSize, dimension, distribution,
                                        tempCounter+1, total_time, LU_time);
  filename := format('/tmp/test_%s_%s_%s_%s_tableSizes.csv', sampleSize,
                           dimension, distribution, tempCounter+1);
  qry := FORMAT('COPY (SELECT * FROM

  (SELECT
  relname as "Table",
  pg_size_pretty(pg_total_relation_size(relid)) As "Size",
  pg_size_pretty(pg_total_relation_size(relid) - pg_relation_size(relid))
  as "External Size"
  FROM pg_catalog.pg_statio_user_tables
  ORDER BY pg_total_relation_size(relid) DESC)
  AS tempAlias ) TO %L CSV HEADER', filename);

  EXECUTE qry;
END;
$$ LANGUAGE plpgsql;
```

Listing 13: testFunc.sql

```
CREATE OR REPLACE FUNCTION test1Code3()
RETURNS void AS $$
```

```
BEGIN
  FOR i in 1..3 -- make 3 tests
  LOOP
    FOR expN in 2..6
    LOOP
      PERFORM testfunc(1000000,8);
    END LOOP;
  END LOOP;
END;
$$ LANGUAGE plpgsql;
```

Listing 14: test1Code3.sql

## B.4 Codes for $\beta = 1$ Tests

```
CREATE OR REPLACE FUNCTION testBeta1(sampleSize INTEGER, dimension INTEGER)
RETURNS void AS $$
/*
*  Function to test correctness of the algorithm and computer precision
*/
DECLARE
  tempBeta DOUBLE PRECISION;
  tempError DOUBLE PRECISION;
  maxError DOUBLE PRECISION;
  tempCounter INTEGER;
BEGIN
  perform testFuncBeta(sampleSize, dimension, 2, 1, 100);
  maxError:=0.0;
  for i in 1..dimension+1
  LOOP
    SELECT beta_val INTO tempBeta FROM beta WHERE beta_row=i;
    tempError := ABS(1-tempBeta);
    IF tempError > maxError THEN
      maxError := tempError;
    END IF;
  END LOOP;
  SELECT count(*) INTO tempCounter FROM betaequals1error
  WHERE b1_sampleSize = sampleSize AND b1_dimension = dimension;
  INSERT INTO betaequals1error
  VALUES(sampleSize, dimension, tempCounter, maxError);
END;
$$ LANGUAGE plpgsql;
```

Listing 15: testBeta1.sql

```
CREATE OR REPLACE FUNCTION testFuncBeta(sampleSize INTEGER, dimension INTEGER,
  distribution INTEGER DEFAULT 1, minValue DOUBLE PRECISION DEFAULT 0.0,
  maxValue DOUBLE PRECISION DEFAULT 1.0)
RETURNS void AS $$
/*
*  Function to test correctness of the algorithm and computer precision
*  Distribution:          1  Uniform, 2  Constructed (beta=1)
*  minValue, maxValue: Define interval for random numbers
*/
DECLARE
  tempCounter INTEGER;
  LU_time INTERVAL;
BEGIN
  -- Initialization:
  PERFORM CreateTables();
  PERFORM createRandomTable(sampleSize, dimension,
    distribution, minValue, maxValue);

  -- Find linear regression coefficients:
  select linReg(dimension) INTO  LU_time from dataSet;
END;
$$ LANGUAGE plpgsql;
```

Listing 16: testFuncBeta1.sql

```
CREATE OR REPLACE FUNCTION runBeta1Test()
RETURNS void AS $$
BEGIN
  FOR i in 1..10
  LOOP
  PERFORM testBeta1(100, 32);
  END LOOP;
END;
$$ LANGUAGE plpgsql;
```

Listing 17: runbeta.sql

# C Matlab Code

The plots shown in Figure 4, Figure 5 and Figure 6 were created in MATLAB, R2014a (8.3.0.532).

```
[~, ~, ~, ~, lu_h, ~, lu_m, ~, lu_s, ~, tot_h, ~, tot_m, ~, tot_s] =
  textread('/tmp/resTable20140609.txt', '%f%c%f%c%f%c%f%c%f%c%f%c%f%c%f', -1);
cnt=1;
totaltime = zeros(6, 4);
for i = 1:6
    totaltime(i,1) = tot_h(cnt)*3600 + tot_m(cnt)*60 + tot_s(cnt);
    cnt=cnt+1;
end
for i = 1:6
    totaltime(i,2) = tot_h(cnt)*3600 + tot_m(cnt)*60 + tot_s(cnt);
    cnt=cnt+1;
end
for i = 1:6
    totaltime(i,3) = tot_h(cnt)*3600 + tot_m(cnt)*60 + tot_s(cnt);
    cnt=cnt+1;
end
for i = 1:6
    totaltime(i,4) = tot_h(cnt)*3600 + tot_m(cnt)*60 + tot_s(cnt);
    cnt=cnt+1;
end

lutime = zeros(6, 4);
cnt=1;
for i = 1:6
    lutime(i,1) = lu_h(cnt)*3600 + lu_m(cnt)*60 + lu_s(cnt);
    cnt=cnt+1;
end
for i = 1:6
    lutime(i,2) = lu_h(cnt)*3600 + lu_m(cnt)*60 + lu_s(cnt);
    cnt=cnt+1;
end
for i = 1:6
    lutime(i,3) = lu_h(cnt)*3600 + lu_m(cnt)*60 + lu_s(cnt);
    cnt=cnt+1;
end
for i = 1:6
    lutime(i,4) = lu_h(cnt)*3600 + lu_m(cnt)*60 + lu_s(cnt);
    cnt=cnt+1;
end
% ------------------------------------------
% bar plot 2d lu time
% ------------------------------------------
figure;
```

```
bar2d = transpose(lutime(2:6,:)).^(1/3);
hBar=bar(bar2d, 'grouped');
grid on;
colormap(summer);
set(gca, 'XTickLabel', {'8','16','32','64'});
xlabel('Dimension d');
ylabel('Time in (sec\^(1/3))');
title('Time-consumption for LU-part');
legend('n=1.0e+02','n=1.0e+03','n=1.0e+04','n=1.0e+05','n=1.0e+06',
        'Location', 'NorthWest');
% ------------------------------------------
% balkendiagramm total time einzeln, fix d
% ------------------------------------------
t = transpose(totaltime);
figure;
hBar=bar(t(1,2:6).^(1/3));
set(gca, 'XTickLabel', {'2', '3', '4', '5', '6'});
xlabel('Sample size log_{10}(n)');
ylabel('Total time (sec\^(1/3))');
title('Total time-consumption for d=8');
colormap(summer);
grid on;
hold on;

figure;
hBar=bar(t(2,2:6).^(1/3));
set(gca, 'XTickLabel', {'2', '3', '4', '5', '6'});
xlabel('Sample size log_{10}(n)');
ylabel('Total time (sec\^(1/3))');
title('Total time-consumption for d=16');
grid on;
colormap(summer);

figure;
hBar=bar(t(3,2:6).^(1/3));
set(gca, 'XTickLabel', { '2', '3', '4', '5', '6'});
xlabel('Sample size log_{10}(n)');
ylabel('Total time (sec\^(1/3))');
title('Total time-consumption for d=32');
grid on;
colormap(summer);

figure;
```

```
hBar=bar(t(4,2:5).^(1/3));
set(gca, 'XTickLabel', {'2', '3', '4', '5'});
xlabel('Sample size log_{10}(n)');
ylabel('Total time (sec\^(1/3))');
title('Total time-consumption for d=64');
grid on;
colormap(summer);
```

Listing 18: plots.m

# References


[1] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M Hellerstein, and Caleb Welton. MAD Skills : New Analysis Practices for Big Data. 2009.

[2] Marcel Dettling. Applied statistical regression. (Lecture notes for 'Applied Statistical Regression', ETH Zurich, AS 2013), 2013.

[3] David Kernert, Frank Köhler, and Wolfgang Lehner. Bringing Linear Algebra Objects to Life in a Column-Oriented In-Memory Database. pages 1–4, n.d.

[4] Markus Neumann. Multiple linear regression in databases. (Facharbeit im Nebenfach Informatik, Departement of Informatics, University of Zurich), 2014.

[5] Carlos Ordonez. Building statistical models and scoring with UDFs. *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1005–1016, 2007.

[6] PostgreSQL. Postgresql manual 9.3 user-defined aggregates, 2014. [Online; accessed 9-July-2014].

[7] Stefan Sauter. Numerische lineare algebra. (Lecture notes for 'Numerische Lineare Algebra', University of Zurich, FS 2014), 2014.

[8] S. Weisberg. *Applied Linear Regression*. Wiley Series in Probability and Statistics. Wiley, 2nd edition, 1985.