Department of Informatics, University of Zürich

BSc Thesis

QR decomposition integration into PostgreSQL

Toni Pešut

Matrikelnummer: 09-915-513

Email: toni.pesut@uzh.ch

November 12, 2015

supervised by Prof. Dr. M. Böhlen and O. Dolmatova

University of Zurich[UZH]

Department of Informatics

Acknowledgments

Most of all, I want to thank my supervisor Oksana Dolmatova for being so helpful and supportive, and for the feedback and guidance she provided. Also, I want to thank Prof. Dr. Michael Böhlen, head of the database technology group at the University of Zürich, for providing the challenging and interesting topic for this bachelor thesis.

Abstract

There is a great demand for storing and analyzing data inside a database management system (DBMS), as scientists in a wide variety of fields rely on DBMSs as a stable and efficient way of handling data. Currently most analyzing is done by exporting from the DBMS into a statistics program, e.g. R, and importing the solutions. However, the more practical solution would be to have statistical operations available inside the DBMS.

Since QR-Decomposition is a frequently used operation in statistics, the goal of this bachelor thesis is an integration of the QR-Decomposition into PostgreSQL. It has been implemented using the Gram Schmidt algorithm and tested to determine the scalability regarding different table sizes.

Zusammenfassung

In der heutigen Zeit gibt es eine steigende Nachfrage nach Möglichkeiten um Daten in Datenbanksystemen (DBS) speichern sowie analysieren zu können. So setzen Wissenschaftler verschiedener Wissenschaftsdisziplinen vermehrt auf Datenbanksysteme als solide und effiziente Lösung, um Daten zu verarbeiten. In der gegenwärtigen Praxis werden oftmals Daten aus dem DBS exportiert, um es in ein Mathematik- oder Statistikprogramm zu importieren, und die Resultate wieder in das DBS zu importieren. Allerdings wäre es praktischer, würde das DBS selbst diese Funktionen bereitstellen.

Aufgrund der Tatsache, dass die QR-Zerlegung eine häufig benutzte Operation der Statistik ist, wird in der vorliegenden Bachelorarbeit ebendiese in das Datenbanksystem PostgreSQL integriert. Zur Implementierung wurde das modifizierte Gram-Schmidt Verfahren verwendet. Zudem wurde auf Skalierbarkeit für verschiedene Tabellengrössen getestet.

Contents

List of Figures

List of Tables

1. Introduction

1.1. Motivation

Database management systems are an efficient and practical way to deal with the ever increasing amounts of data in scientific environments. For a long time DBMSs have offered great functionality, especially for businesses. However, analysis of scientific data heavily relies on linear algebra operations on matrices and arrays. In order to meet these new demands, DBMSs like MonetDB with SciQL, SciDB or RasDaMan have emerged, which offer such functionalities. Each DBMS of course has its advantages and disadvantages, always depending on the use case. For this thesis, PostgreSQL was chosen. The goal of the thesis is to expand it by integrating one of the most used linear algebra operations, the QR decomposition.

The current approach when linear algebra operations, like e.g. the QR decomposition, are used is to export the data from PostgreSQL to a math or statistics program, like MATLAB or R, and import the results. Obviously it would be more practical to be able to perform these operations directly in the DBMS, without having to leave the database environment.

Of course, there are certain properties of databases that may be a disadvantage for such computations. An important aspect of this is the issue of fast access to rows and columns in a database. This issue and others have to be taken into consideration and the solutions to overcome these will be discussed in this paper. To asses the usability of the implementation, there will also be a performance evaluation to determine the scalability and applicability for matrices of different sizes.

1.2. Problem statement

The increasing amount of data that is collected and processed in scientific environments leads to new demands for DBMSs. The analysis of said data is heavily based on linear algebra operations and requires matrix and array operations. Nowadays, the most common way is to use the DBMS as a storage and outsource those computations to a programs like R and MATLAB, which offer such mathematical and statistical functionalities. A more convenient way to use important and frequently used operations on matrices would be to integrate them in the DBMS, removing the need to export the data to an external program and re-import the data.

Therefore, the goal of this thesis is to integrate one of the most used linear algebra operations, the QR decomposition, into the DBMS PostgreSQL. It serves as an example to assess the feasibility and to determine obstacles for integrating array or matrix based

operations.
The first part of the implementation is the computation of table Q, which includes an optimization to improve the runtime by reducing the number of table scans. The second part consists of the implementation for table R.
To assess the scalability regarding different table sizes, the runtimes are evaluated and presented.

1.3. Thesis Outline

At the start of this thesis, in chapter 2, I will present related work that has already been done, which will include the current possibilities for QR decomposition in PostgreSQL and other linear algebra operations in other DBMSs. Chapter 3 introduces and explains the modified Gram Schmidt algorithm for QR decomposition and its uses. In chapter 4, my solution for an implementation in PostgreSQL is presented. First, in section 4.1, I will define a proposed SQL syntax and explain the reason for dividing the implementation into two parts, namely one for result matrix Q in section 4.3 and one for resulting matrix R in 4.4. Additionally, section 4.3.2 explains the optimizations done for the computation of matrix Q. In the following chapter 5 the experiments done with this implementation are presented. They examine the scalability and runtimes of the implemented algorithm. Finally, in chapter 6 I will summarize the thesis and reflect on the implementation and chances for improvement.

2. Related Work

2.1. Using math and statistics programs with a DBMS

In this chapter different ways of using statistics in databases are presented.
R is a comprehensive environment for statistical computing. It provides a language, called R and includes graphics. R offers a great coverage of statistical operations, including the QR decomposition and is extensible with packages. [1] This makes R an interesting choice for the use with a DBMS. Oracle R Enterprise is a solution, that enables the use of an Oracle Database with R. Oracle R Enterprise consists of three parts. Firstly, there is the "Client R Engine", which provides an interface to R, which can access and modify data on the Oracle Database it is connected to. It does this by using R commands in the client and translating them to SQL, which is then sent to the server. After the computation is done, the database server sends back the results to the client. The second actor is the Oracle Database server. It is an extended version of the Oracle Database, which contains additional functionality in order to be able to execute the broadened functionalities introduced by the R client. The third part are the R Engines, which can be spawned by the database to provide additional capabilities like scheduling, triggering scripts etc. There is a broad range of linear algebra operations that are available; e.g. it is possible to build a regression model [2]

A different possibility to use statistics with Oracle is to use database packages. The UTL_NLA package for Oracle Database provides linear algebra functionality. It provides operations on matrices and vectors, which are stored in a data type called VARRAY. The operations available are quite comprehensive. This includes operations which will be introduced in this paper such as solving linear systems, and also utilizes functions which use QR decomposition. [3]

2.2. Linear algebra operations in database management systems

In this section I will present existing DBMSs and query languages which offer extended support for linear algebra and matrix operations compared to standard SQL database systems.

In an effort to facilitate array computations in a DBMS, SciQL, a SQL based query language was introduced. The main goals were to simplify and optimize array operations in SQL and to enable working with table as well as array types in order to maintain the ability to work with existing software libraries. SciQL's big innovation is the introduction

of an ARRAY type, which works on the same level as a TABLE. In practice this means, that an ARRAY is not stored as a data field in a TABLE, but is an entity of itself. The main structural difference between a TABLE and an ARRAY is that the former consists of a set of tuples, while the latter consists of cells, which are basically tuples with indices. In SciQL it is possible to use arrays instead of tables in a SQL expression. A further advantage of SciQL is that ARRAY syntax in general is largely based on table syntax in SQL, making the transition from SQL to SciQL easier. Another strong point is that it easy to switch between TABLE and Array, via SELECT statements with indices or explicit casting. [12]

However, there are some disadvantages to SciQL.For one, relational algebra, on which SQL is based, is not applicable to arrays. This is a big issue, as it shows that it is not possible to add ARRAY as a base type on the same level as TABLE, because the relational algebra does not define operators on arrays, but is defined for relations (tables). As such the ARRAY type can not be considered equal to the TABLE type.
Another drawback is the overhead created by the indices, especially when using big arrays. If e.g. the index is of an integer type, which would make sense in a lot of cases, the memory consumption will be increased, as a lot of indices would be needed for such a big table.

SciQL was implemented using MonetDB as a back end. First, a SciQL query is parsed, which generates a syntax Tree. This tree is compiled by the SciQL query compiler, which has MonetDB assembly language as a target language. To sum up, SciQL has managed to offer a much improved usability and efficiency when using complex SQL expressions on arrays. [13]

Another approach was taken by the developers of SciDB. One of the main gripes with using existing DBMSs was the lack of an appropriate data model, inadequate capabilities, and a general doubt if these issues will be addressed. It appears that most database vendors are more focused on business market. Thus, SciDB was developed. Unlike SciQL, SciDB was not built on top of a existing DBMS. It is an open source DBMS, built specifically in order to meet the scientist's demands for supporting complex analytics, array data and working with large data sets (see [10] for further requirements). Based on the requirements of the data used in this scientific contex, an array data model was adopted. This decision was also based on the fact that the required analytics commonly use linear algebra operations. It also eliminates the need for conversion between tables and arrays. Another notable feature is the ability to use nested, multi-dimensional arrays. As for the query and functional languages used, SciDB introduces a query language called AQL, which is heavily based on SQL, and a functional query language called AFL. In order to enable efficient handling of huge datasets (e.g. in astronomy), SciDB is designed to run on a grid of computers. This is an advancement for this use case, as a lot of statistical tools (e.g. R), which were used in conjunction with "traditional" relational DBMSs, rely on computations on a single node (computer), which is not feasible for such huge datasets. What is more, they only offer analytics without data management and require exporting the data from the DBMS, analyzing it, and reimporting it, as well as

conversions from tables to matrices and from matrices to tables. [10]
Some critics say that the concept of using arrays as tables, like in SciDB, is not optimal. When working with millions of these array tables, there is no convenient way to iterate over a whole set of tables, as a relational database was not intended to be used with these huge numbers of tables. Also, the users have to have the rights to modify schemas in order to be able to effectively work with an array based DBMS, which might be problematic from a security standpoint. [8] However, this applies to all approaches presented here (apart from ASQLDB). E.g. in PostgreSQL, when using a table to store an array, there is also exactly one table for each array, introducing the same problems. The user also has to have the rights to modify schemas, otherwise it is not possible to use arrays in tables.

The final alternative presented here is ASQLDB, built on top of the array DBMS RasDaMan and relational DBMS HSQLDB.ASQLDB responds to the need for array data processing by extending the standard SQL language and introduces Array SQL (ASQL) as a new query language. As already mentioned, is uses a relational DBMS together with a relational DBMS, which sets it apart from the previously mentioned solutions. The concept behind using an array DBMS in conjunction with a relational DBMS is that a pure array DBMS implementation was deemed lacking, as the data itself without any metadata is not very meaningful. The workaround in other systems was linking the array data with the metadata, which is not an elegant solution in a array DBMS. In ASQL, the problem was adressed by adding arrays as an attribute type, meaning that the array is stored in a table and can not be stored as an entity by itself, like in SciQL or SciDB. Depending on the use case, introducing arrays as attributes can have advantage over arrays as tables. To provide an example, it is not defined by SQL how to iterate over whole sets of tables. If the arrays are stored in a table however, it is in line with the standard way of working with SQL. The query language offers many array operations, like aggregation, subsetting, extending, shifting etc. The ability to directly manipulate arrays facilitates the introduction of user defined functions. [8]

To summarize the presented DBMSs, I have included table 2.1, which shortly summarizes the main differences.

DBMS	PostgreSQL	SciQL(MonetDB)	SciDB	ASQLDB (RasDaMan)
Query Language	SQL	SciQL	AQL	ASQL
Function language	SQL	SciQL	AFL	ASQL
underlying data structure	table	table & array	array	table (arrays as attributes)
linear algebra operations	partially built-in	partially built-in	built-in	built-in
expanding lin.alg. capabilities	UDF or extension of PostgreSQL core	user-defined functions	user-defined functions	user-defined functions

Table 2.1.: Comparison between the DBMSs

As can be seen, PostgreSQL does offer some linear algebra operations. This includes operations like average, minimum, maximum, sum etc. It is limited however, as operations as e.g. matrix multiplication are not built-in. The same holds true for SciQL, where an operation as e.g. matrix inversion is not built-in.
There is still the possibility to expand all DBMSs by introducing user-defined functions for more specific or complex linear algebra operations. Since PostgreSQL does not inherently offer a standard way of working with matrices (e.g. no direct access to "cells via matrix indices) it might be necessary to extend the PostgreSQL core and introduce new keywords for these functions.

I want to give some insight into why PostgreSQL was chosen for the integration of the QRD in this paper.
Of course, each option has its advantages. When working with huge amounts of data on a grid of servers, SciDB can be very powerful. However this is rather the case in large institutions, e.g. in astronomy. In this case, we are dealing with much smaller sets of data however, and it will also only be used on a single computer. QRD is suitable for the integration in PostgreSQL, because it allows a solution, which works with a minimal overhead in the table structure. Only an additional index to identify the rows is needed (this will be explained later in 4.2); the column order is fixed anyways. Besides, it allows introducing the linear algebra capabilities within an existing infrastructure, without the need to migrate to run different databases in parallel or learning a new query language.

3. QR decomposition

3.1. Definition of QR decomposition

In general, QR decomposition, or QR factorization as it is also called, is a procedure where a given Matrix $\boldsymbol{A}$ is decomposed into two matrices $\boldsymbol{Q}$ and $\boldsymbol{R}$. We assume that $\boldsymbol{A} \in \mathbb{R}^{(m \times n)}$ with $rank(\boldsymbol{A}) = n$ and $m \geq n$, where m denotes the number of rows and n the number of columns.
The decomposition fulfills the equation:

$$\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{R} \tag{3.1}$$

where matrix $\boldsymbol{Q}$ is orthogonal ($\boldsymbol{Q}^{\boldsymbol{T}}\boldsymbol{Q} = \boldsymbol{I}_n$) with dimensions $m \times n$ and $rank(\boldsymbol{Q}) = n$, and $\boldsymbol{R}$ is a right upper triangular matrix. These properties are useful for the applications seen in section 3.3. [6]
Now let us look at an example.

Matrix $\boldsymbol{A} := \begin{bmatrix} 1.3 & 4.7 & 4.9 \\ 0.1 & 3.3 & 2.8 \\ 3.8 & 4.3 & 1.6 \\ 1.2 & 3.9 & 4.8 \\ 9.0 & 4.7 & 3.9 \end{bmatrix}$ yields the resulting matrices (rounded to 3 decimal places):

$$\boldsymbol{Q} = \begin{bmatrix} 0.131 & 0.602 & 0.207 \\ 0.010 & 0.514 & -0.155 \\ 0.383 & 0.257 & -0.806 \\ 0.121 & 0.486 & 0.470 \\ 0.906 & -0.266 & 0.250 \end{bmatrix} \text{ and } \boldsymbol{R} = \begin{bmatrix} 9.929 & 7.026 & 5.397 \\ 0.0 & 6.277 & 6.098 \\ 0.0 & 0.0 & 2.519 \end{bmatrix}$$

3.2. Definition of the modified Gram Schmidt algorithm

To calculate the matrices $\boldsymbol{Q}$ and $\boldsymbol{R}$ the modified Gram Schmidt algorithm was chosen and will be introduced in the following section.
The input of the modified Gram Schmidt algorithm (MGS) is matrix $\boldsymbol{A}$ ($m \times n$) and the two resulting matrices are $\boldsymbol{Q}$ and $\boldsymbol{R}$. In the algorithm definition the notation $\boldsymbol{a}_{jk}$ refers to the element of $\boldsymbol{A}$ in row j and column k. Algorithm 1 contains the definition of the modified Gram Schmidt algorithm.

Algorithm 1 Modified Gram Schmidt algorithm [6]

```
1:  for k := 1 to n do
2:      s := 0
3:      for j := 1 to m do
4:          s := s + a_jk^2
5:      end for
6:      r_kk := sqrt(s)
7:      for j := 1 to m do
8:          q_jk := a_jk / r_kk
9:      end for
10:     for i := k + 1 to n do
11:         s := 0
12:         for j := 1 to m do
13:             s := s + a_ji * q_jk
14:         end for
15:         r_ki := s
16:         for j := 1 to m do
17:             a_ji := a_ji - r_ki * q_jk
18:         end for
19:     end for
20: end for
```

The algorithm works as follows: m and n refer to the number of rows respectively the columns, as mentioned in the introduction above. The outermost for loop in row 1 iterates over all columns k. Then variable s is initialized to zero and the for loop in row 3 successively adds the square of the elements from $\boldsymbol{A}$ in column k to s. In row 6, the main diagonal elements of $\boldsymbol{R}$ are updated. The next for loop in row 7 updates all elements of $\boldsymbol{Q}$ in column k.
Following that is a loop from row 10 to 19 that updates the remaining columns from $k+1$ until the last column and contains two inner loops. The first inner loop in row 12 instantiates a new variable s. The product of $\boldsymbol{a_{ji}}$ and $\boldsymbol{q_{jk}}$ is successively added to s via an iteration over all rows j. After that element $\boldsymbol{r_{ki}}$ is updated with the value s. Finally, the second inner loop in row 16 updates the whole column vector i of $\boldsymbol{A}$ and the algorithm resumes at row 10. When this loop is finished, the algorithm jumps to row 1 until all columns are computed.

The algorithm has a complexity of $\mathcal{O}(m * n^2)$ [7], as there are three nested for loops, with the outer two iterating to n and the innermost iterating to m.

As we see, in a given step k, $\boldsymbol{A}$'s column vectors $k+1$ to n are updated as well as $\boldsymbol{Q}$'s column vector k, and $\boldsymbol{R}$'s row k starting from the main diagonal element.

3.3. Applications for QR decomposition

One of the applications for QR decomposition is matrix inversion. The task is to compute the inverse of a square matrix A. The computation is facilitated by using the properties of Q and R. Using the fact that Q is square and orthogonal, meaning that $\boldsymbol{Q}^{-1} = \boldsymbol{Q}^T$, and using basic matrix rules we can do the following transformations:

$$\begin{aligned} \boldsymbol{A}^{-1} &= (\boldsymbol{QR})^{-1} \\ &= \boldsymbol{R}^{-1}\boldsymbol{Q}^{-1} \\ &= \boldsymbol{R}^{-1}\boldsymbol{Q}^T \end{aligned}$$

$\boldsymbol{R}^{-1}$ is easier to compute than $\boldsymbol{A}^{-1}$ because $\boldsymbol{R}$ is triangular. [9]

Another application for the QRD is the least squares (LS) problem. Given is a data matrix $\boldsymbol{A} \in \mathbb{R}^{m\times n}$ and a observation vector $\boldsymbol{b} \in \mathbb{R}^m$. Since the equation $\boldsymbol{Ax} = \boldsymbol{b}$ is overdetermined, there is no generally no exact solution. As such, the goal is to minimize $\|\boldsymbol{Ax} - \boldsymbol{b}\|_p$. Although p indicates that different norms can be chosen, the least squares problem here will be using the 2-norm, because multiplying with an orthogonal matrix does not change the 2-norm. As a consequence the problem can be transformed to minimizing $\|(\boldsymbol{Q}^T\boldsymbol{A})\boldsymbol{x} - (\boldsymbol{Q}^T\boldsymbol{b})\|_p$, which is easier to solve. Multiplying the equation $\boldsymbol{A} = \boldsymbol{QR}$ from the left with $\boldsymbol{Q}^T$ results in:

$$\begin{aligned} \boldsymbol{A} &= (\boldsymbol{QR}) \\ \boldsymbol{Q}^T\boldsymbol{A} &= \boldsymbol{Q}^T\boldsymbol{Q} * \boldsymbol{R} \\ \boldsymbol{Q}^T\boldsymbol{A} &= \boldsymbol{R} \end{aligned}$$

So the LS problem is reduced to minimizing $\|\boldsymbol{Rx} - (\boldsymbol{Q}^T\boldsymbol{b})\|_p$ [7]

A further application for the QRD is singular value decomposition (SVD). The SVD decomposes a real matrix $\boldsymbol{A} \in \mathbb{R}^{m\times n}$ with $m \geq n$ such that $\boldsymbol{A} = \boldsymbol{U\Sigma V}^T$ The QR decomposition can be used here for triangulation and can be a step in the computatio of the SVD. The SVD can be used to compute the rank, pseudoinverse, and can be used to solve the LS problem. [7]

The QR decomposition can also be used to determine the numerical rank of a given matrix A. This is done by a slight modification of the QRD. A permutation matrix $\boldsymbol{P}$ is introduced such that $\boldsymbol{AP} = \boldsymbol{QR}$ and

$$R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & R22 \end{bmatrix}$$

with $\boldsymbol{R_{11}} \in \mathbb{R}^{r \times r}$. The goal of the permutation is to yield a "small" $\boldsymbol{R_{22}}$. If it satisfies a specified condition, it can be concluded that matrix $\boldsymbol{A}$ has at most numerical rank r. [11]

4. QR decomposition integration in PostgreSQL

In this chapter I will first explain which challenges I faced while integrating QR decomposition into PostgreSQL and discuss the difference between working with matrices and tables when working with linear algebra algorithms inside of databases. In the following sections I will present my solutions for the computation of $\boldsymbol{Q}$ and $\boldsymbol{R}$.

4.1. Relational algebra & SQL queries for QRD

Firstly, as mentioned in section 3.2 the QR decomposition of Matrix $\boldsymbol{A}$ produces two result matrices $\boldsymbol{Q}$ and $\boldsymbol{R}$. These matrices can be computed simultaneously. However, PostgreSQL is a relational database and in relational algebra the result of an operation can only be a single relation. Thus, the implementation wars subdivided into two parts, a solution for $\boldsymbol{Q}$ in section 4.3 and a separate one for $\boldsymbol{R}$. in section 4.4.

Before the new operations are introduced, it is important to make an assumption. In order for the QRD to work properly, it is absolutely necessary to introduce a rowId. The rowId must be the first attribute in all relations where the QRD should be computed on. It is necessary for the purpose of correctly identifying the row numbers of the input matrix, on which the QRD is executed. Also, the rowId must be unique and should be continuously incremented for every row.

Relational algebra defines basic operators, like select, project, join etc on relations. A schema defines the attributes of the relation.[5]
For the QRD two new operations will be introduced. The first operation is Q_QR. The notation is Q_QR(r), where r is a relation on the schema R(rowId, A_1, .., A_n). The attributes rowId, A_1, .., A_n must be numeric and it is mandatory that rowId is the first attribute in the relation. Q_QR returns the resulting relation q, which is a relation on the same schema R(rowId, A_1, .., A_n) and consists of the computed "Q" of the QRD with relation r as input.
The proposed corresponding SQL command is:

SELECT * FROM Q_QR(r); (where "r" is the table name)

The second operation is R_QR. The notation is analogous to the one above; it is R_QR(r), with r being a relation on the schema R(rowId, A_1, .., A_n). Again, the attributes rowId, A_1, .., A_n must be numeric and rowId must always be the first attribute

of the relation. The operation returns a resulting relation on the same schema R(rowId, A_1, .., A_n) and consists of "R", the upper triangular matrix of the QR decomposition. The proposed SQL command is:

SELECT * FROM R_QR(r);

[5]

Here is an example.

We have an input matrix $\boldsymbol{A} = \begin{bmatrix} 0.5277707161567724 & 13.857163243639882 \\ 23.19361663602909 & 10.723462700064045 \\ 45.50729547213681 & 92.35359509996742 \end{bmatrix}$

The representation as a relation in the database is shown in Table 4.1. Note the added column rowId, which was introduced above, which is needed to identify the row numbers.

rowId	col0	col1
0	0.5277707161567724	13.857163243639882
1	23.19361663602909	10.723462700064045
2	45.50729547213681	92.35359509996742

Table 4.1.: Relation A

Executing the QR decomposition with $\boldsymbol{A}$ as input yields the following results, shown in table 4.2 and 4.3. The values were computed using the presented implementation in this paper.

rowId	col0	col1
0	0.0103322975100351	0.371440055574439
1	0.454067154695955	-0.828949202395153
2	0.890907550004199	0.418181186763939

Table 4.2.: Resulting relation Q

rowId	col0	col1
0	51.0797057134855	87.2908636745764
1	0	34.8784356340102
2	0	0

Table 4.3.: Resulting relation R

4.2. Challenges of computing linear algebra operations in databases

There is a disparity in how a database such as PostgreSQL works compared to the way in which linear algebra operations are usually computed. One aspect is that PostgreSQL is

a tuple-based DBMS. This means that in order to access a certain value in the database the whole row of the relation has to be fetched. What is more, the rows can usually be only accessed sequentially, i.e. to read or modify a value in row 'k' all rows before k have to be iterated through first.The exception to that is the possibility to mark a row and later get back to its position, still meaning that it is not possible to simply traverse "upwards" in the relation.
This of course has implications for the implementation. On one hand the execution is slowed down, when the algorithm calls for only single values of a row and iterates through a whole column, as is the case here (see section 3.2). All rows have to be fetched even though not all values are needed, which produces an unnecessary overhead. As a consequence, it does not make sense to directly access the database each time an entry of the relation is accessed. This led to the implementation of a new data type that stores the values of the relation in a two-dimensional self-implemented array, where the values from the relation are stored.
An array has the advantage, that the cells can be directly accessed using two indices x and y; thus the operations run much faster compared with a solution that relies on directly accessing the data in the table. This is why all implementations to obtain Q or R first read all values into an array.

Another difference is that the order of rows in a relational DBMS in general, not only in PostgreSQL, is not guaranteed [4] (with the exception of statements which use an ORDER BY clause). For the implementation of the MGS this means that a row index must be introduced so that the correct result can be obtained. Otherwise the output rows could be in an arbitrary order. A different row order of $\boldsymbol{A}$ changes $\boldsymbol{Q}$, but leaves $\boldsymbol{R}$ unaffected.
This leads to a fixed schema R(rowId, A_1, .., A_n) that has to be used, with rowId, A_1, .., A_n being numeric and requiring rowId to be the first attribute of the relation. Table 5.2 serves as an example to illustrate the table structure.

rowId	col0	col1	col2
0	1.3	4.7	4.9
1	0.1	3.3	2.8
2	3.8	4.3	1.6
3	1.2	3.9	4.8
4	9.0	4.7	3.9

Table 4.4.: Table structure with rowId

4.3. Implementation for table Q using the modified Gram Schmidt algorithm

In this chapter I will present my implementation of the QR decomposition using the modified Gram Schmidt algorithm. In a first step, I will present an unoptimized solution

for the matrix Q, where the input matrix A is scanned each time for every row of the resulting relation. In the following subsection an optimized solution is presented, where tuple fetches from the database are minimized in order to improve the execution time. In 4.4 the solution for the matrix R is presented.

In listing 4.1, the C code for the QRD can be seen. When comparing the C implementation with the algorithm definition in 3.2, two things become obvious. Firstly, it is evident that the column and row indices are interchanged. This is owed to an optimization that was done in the way the data is stored and will further be explained in section 4.3.2. Secondly, the interchanged indices do not match up with the original algorithm when changed back again. This is owed to the fact that a rowID was introduced in 4.2 because the DBMS does not guarantee row order and has no further impact on the algorithm, as the indices are simply copied to the resulting relation.

```
void QR(){
  // init result arrays
  initArray(&q, a.sizeX, a.sizeY);
  initArray(&r, a.sizeX, a.sizeY);
    //start QR decomposition
    int k;
    int indexOffset = 1;
    for (k = 1 + indexOffset; k <= a.sizeY; k++) {
      double s = 0.0;
      int j;
      for (j = 1; j <= a.used; j++) {
        double temp;
        temp = a.array[k - 1][j - 1] * a.array[k - 1][j - 1];
        s = s + temp;
      }
      r.array[k - 1][k - 2] = sqrt(s);
      for (j = 1; j <= a.used; j++) {
        q.array[k - 1][j - 1] = a.array[k - 1][j - 1]
            / r.array[k - 1][k - 2];
      }
      int i;
      for (i = k + 1; i <= a.sizeY; i++) {
        double s2;
        s2 = 0.0;
        int j;
        for (j = 1; j <= a.used; j++) {
          s2 = s2 + a.array[i - 1][j - 1] * q.array[k - 1][j - 1];
        }
        r.array[i - 1][k - 2] = s2;
        for (j = 1; j <= a.used; j++) {
          a.array[i - 1][j - 1] = a.array[i - 1][j - 1]
              - r.array[i - 1][k - 2] * q.array[k - 1][j - 1];
        }
      }
    }
    //copy row indices to result tables
    int ind;
```

```
    for (ind = 0; ind < a.used; ++ind) {
39    q.array[0][ind] = a.array[0][ind];
      r.array[0][ind] = a.array[0][ind];
41  }
}
```

Listing 4.1: QR decomposition using modified Gram Schmidt implementation in the Postgres backend

For all the implementations for Q and R it is important to know some basic information about the input and output parameters.

The input of the PostgreSQL executor is a plan tree created by the planner/optimizer, which is recursively processed. The method called for returning the resulting tuple is fed with such a node until the last row is reached.[4]
The node contains all data and meta information needed for the execution such as the the execution plan and its attributes as well as runtime variables, such as the current node with its associated data, a storage for the resulting tuple for the current node, among other information needed for the execution of the operation.

4.3.1. Brute force implementation

In listing 4.2 the entry point of the brute force implementation for table R can be seen. As already mentioned, the input value node represents the current node in the plan tree which is executed.

The drawback of using a DBMS for operations on arrays is that always a whole row is fetched at once. There is no way to fetch a single value from a row, which would be useful for a faster execution of the algorithm. Based on this consideration I introduced a new data structure to hold all the values of input relation A. The values are stored in a dynamic two dimensional array and the rows and columns can thus neatly be accessed and operated on with commonly used matrix indices.

This code excerpt shows how the entire input relation, from here on refered to as A, is scanned and stored in such a temporary array.

First, it is important to store a reference to the first tuple of A. This is needed in order to be able to get access to the whole input table. Otherwise it would not be possible e.g. to access row k-1 when the current node is at position k. This is needed, as can be seen in the algorithm definition of the QRD (see Algorithm 1 in 3.2). Any given row k's result for the QRD depends on all rows of A. After this, a temporary array is initialized which stores the contents of table A by successively fetching tuples and storing them. When the last row of A is reached, the QR decomposition can be executed and the loop is exited.

```
// Brute force implementation; Scans the whole relation for every output
   row
2 TupleTableSlot // return: a tuple or NULL
ExecUnique_Q_QR_bruteforce(UniqueState *node) {
4   // runtime variables and fetching node information
    Unique *plannode = (Unique *) node->ps.plan;
```

```
6     TupleTableSlot *resultTupleSlot;
      TupleTableSlot *slot;
8     PlanState *outerPlan;
      ...
10  // array which stores the values of the current row
    double *currentRow;
12  // mark the position in the plan for later restore
    if (currentIndex == 0) {
14    ExecMarkPos(outerPlan);
    }
16
    //restore Position to the first node (needed for the next row)
18  ExecRestrPos(outerPlan);
    outerPlan = outerPlanState(node);
20  resultTupleSlot = node->ps.ps_ResultTupleSlot;

22  // fetch whole input table, store data & execute QR decomposition
    for (;;) {
24    // fetch tuple
      slot = ExecProcNode(outerPlan);
26    // init array storing input table values
      if (a.array == NULL) {
28      initArray(&a, 10, plannode->numCols);
      }
30    // reached last tuple -> call QR decomposition algorithm
      if (TupIsNull(slot)) {
32      QR();
        break;
34    }
      int numColumns = plannode->numCols + indexOffset;
36
      // allocate space for values of current row
38    currentRow = (double *) malloc(numColumns * sizeof(double));
      // fetch attributes of the current row and store them in the array
40    ...
      // insert current row into the temporary array
42    insertRow(&a, currentRow);
    }
```

Listing 4.2: Table scan brute force

After this, the position is restored to the current row in order to return the correct row as a result. It has to be checked, if the last input row has already been reached. If it has been reached, the necessary clean up methods are called, which clear any allocated memory as well as runtime variables, and NULL is returned to signal that all output rows have been returned. If the last row is not reached yet, the previously calculated values for the current row are fed into the structure holding the values that are returned. Then the current row is returned and the end of the method is reached. The whole QR decomposition is completed when all incoming nodes from the plan tree have been processed.

4.3.2. Optimized implementation for matrix Q

The idea behind the optimized implementation of the Gram Schmidt algorithm is to minimize the number of times the data from the input table has to be read from the database.

As in the brute force solution, the input table A has to be scanned and stored in a temporary array. During the first execution, the first tuple is fetched and the array A is initialized. The next step is the iteration through the values of the current row which are stored in the array afterwards. The break conditions of the loop or reached if either the last tuple is reached (which means that the input row is empty) or when the input table has already been wholly stored to the array already. Subsequent executions of the method do not execute any database fetches, as the data needed is already read into the memory. If the last tuple was reached, the QR decomposition is executed. The QR decomposition using the Gram Schmidt algorithm is only called once and computes matrix Q. The result is stored in array q.

As a next step the position is restored to the current row and the second for loop is entered (see listing 4.3). This loop contains clean up methods, which are executed when the last row is reached; the allocated arrays are freed, the row index is reset and NULL is returned, so that the method which called ExecUnique knows, that the last tuple was already returned. If the last row has not been reached yet, the ResultTupleSlot is filled with the results from the QR decomposition.

```
ExecUniqueWithRowIndex(UniqueState *node){
...

  // restore the position in the plan for returning the resulting relation
  ExecRestrPos(outerPlan);
  outerPlan = outerPlanState(node);
  resultTupleSlot = node->ps.ps_ResultTupleSlot;
  for (;;) {

     // fetch a tuple from the outer subplan

    slot = ExecProcNode(outerPlan);

    if (TupIsNull(slot)) {
      // reset variables to be able to run QR Algorithm again
      reachedLastRow = false;
      freeArray(&q);
      freeArray(&r);
      currentIndex = 0;
      freeArray(&a);
      // end of subplan, so we're done
      ExecClearTuple(resultTupleSlot);
      return NULL;
    }
    break;
  }
  ExecClearTuple(resultTupleSlot);
  // take QRD result row and store it in the result tuple
  int i;
  for (i = 0; i < slot->tts_tupleDescriptor->natts; i++) {
    resultTupleSlot->tts_values[i] = Float8GetDatum(
        q.array[i][currentIndex]);
  }
  // set flags, to mark the result containing data
  for (i = 0; i < slot->tts_tupleDescriptor->natts; i++) {
    resultTupleSlot->tts_isnull[i] = false;
  }
  ++currentIndex; // update row index
  // mark tuple as virtual and return
  return ExecStoreVirtualTuple(resultTupleSlot);
}
```

Listing 4.3: excerpt of the optimized implementation for Q

This implementation enables us to fetch each row from the database only once and improves the performance twofold. Firstly, the table only has to be read from the database once. Reading from the database is a costly operation and as such this improves the runtime greatly. Secondly, the QRD only has to be executed once. After the first row, all subsequent rows merely have to fetch the result from the memory without having to calculate the result values first.

A further optimization was done on the way the arrays are stored in the memory. Based on the algorithm definition in 3.2 it can be seen that the algorithm's complexity is linear with regards to the number of rows, but grows in a quadratic fashion when increasing column numbers. When examining the algorithm it can also be seen that the innermost for loops iterate over all rows, while the column is fixed. Also the two loops in line 3-9 iterate over all rows. Iterating over rows means, that the matrices are read from "top" to "bottom". This led to the decision, that the focus of the data structure that holds table A (as well as Q) should be to provide fast access for reading columns, as this improves the speed when iterating over rows. This is achieved by transposing the matrices stored in the memory, and is also the reason why the indices in the QR decomposition code in 4.3 are interchanged. Of course, the result relation that is returned has the expected dimensions and is not transposed.

Here is an example to illustrate the issue.
Given a Matrix $A = \begin{bmatrix} 0 & 7 & 8 \\ 1 & 9 & 10 \\ 2 & 5 & 11 \end{bmatrix}$ the task is to get a sequential access over columns, which would speed up the Gram Schmidt algorithm used for the QRD, as introduced in section 3.2.
The ordinary way to store the array would have a sequential access when going to the right in the same row, so there would be the arrays:

$$A_1 = \begin{bmatrix} 0 & 7 & 8 \end{bmatrix}, A_2 = \begin{bmatrix} 1 & 9 & 10 \end{bmatrix} \text{ and } A_3 = \begin{bmatrix} 2 & 5 & 11 \end{bmatrix}$$

In this notation the index next to A defines the row of the original matrix. It is evident that sequential access can only be granted when traversing through a row.
Using a transposed two-dimensional array to store the matrix, we get the following arrays in the memory (the indices 1-3 refer to the column):

$$A_1 = \begin{bmatrix} 0 & 1 & 2 \end{bmatrix}, A_2 = \begin{bmatrix} 7 & 9 & 5 \end{bmatrix} \text{ and } A_3] = \begin{bmatrix} 8 & 10 & 11 \end{bmatrix}$$

This solution provides the desired sequential access over columns, that accelerates the computation of the QRD.

4.4. Implementation for returning the table R using the Gram-Schmidt algorithm

As can be seen in chapter 3.2, in listing 4.1, the QR() method is implemented in such a way, that the Q as well as the R matrix are calculated. Therefore the implementation for the table R only slightly differs from the solution presented in section 4.3.2. The difference being, that in listing 4.3, line 32, the values that are to be returned are taken from the r array, not from the q array. As already explained in 4.3.2, the arrays q and r have already been filled with the values from the QRD at this point.

5. Experiments

In this chapter the data gathered while testing the implementation is presented.

The test system was a Lenovo T400, equipped with a dualcore Intel Core2Duo T9600 running at 2.8 GHz, 4 GB of RAM and has a 240 GB SSD. It is running a 64-bit version of Ubuntu 14.04 LTS.

The experiments were divided in two parts. In one part, a fixed column size was used, and row were added subsequently to assess the scalability regarding the number of rows. There is also a comparison between the runtimes of the brute force solution and the optimized solution. In the other part a fixed number of rows and a varied number of columns were used. The runtimes depicted are the averages of three runs.

Table 5.1 and 5.2 contain the results for the execution of the operation Q_QR. For this test, the column number remained at 100, and the row numbers were steadily increased.

Number of rows	Number of columns	Execution time in s
100	100	0.168
500	100	0.750
1'000	100	1.49
5'000	100	7.47
10'000	100	14.97
50'000	100	87.87
100'000	100	175.19
500'000	100	879.89

Table 5.1.: Runtimes optimized Q_QR, fixed column number

Number of rows	Number of columns	Execution time in s
100	100	0.296
500	100	6.502
1'000	100	25.989
5'000	100	109.878
10'000	100	aborted after 20 minutes

Table 5.2.: Runtimes brute force Q_QR, fixed column number

Figure 5.1 compares the execution time of the optimized and the brute force solution. Note that the brute force solution did not terminate for 10'000 rows in the time the optimized solution computed 500'000 rows. Therefore the execution was halted.

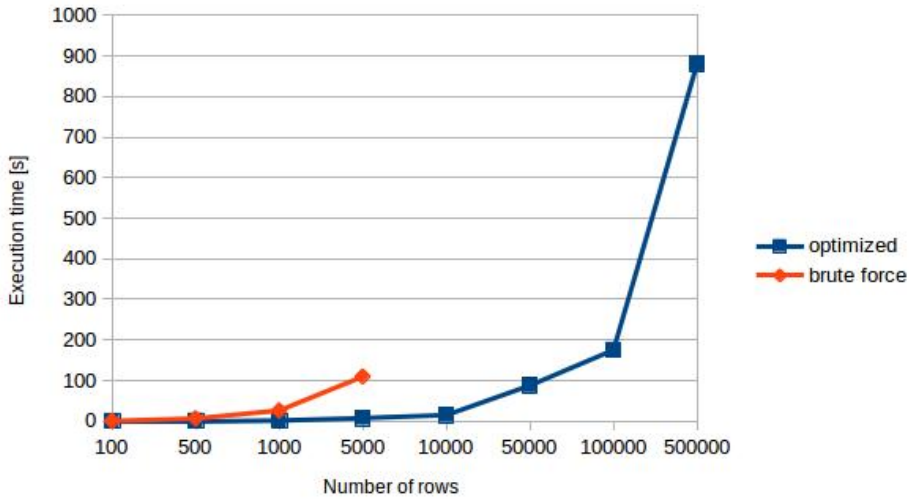

Figure 5.1.: Execution times for Q_QR with a fixed column number of 100

It can be seen, that the execution time for the optimized Q_QR implementation behaves quite linearly when increasing the row numbers. This is in line with the expectation. In section 3.2 the QRD algorithm was shown to have a complexity of $\mathcal{O}(m * n^2)$, with m denoting the number of rows and n the number of columns.

The following results are from the experiment with a fixed row number of 1000 and varying column numbers.

Number of rows	Number of columns	Execution time in s
1'000	10	0.152
1'000	25	0.372
1'000	50	0.742
1'000	100	1.493
1'000	250	5.276
1'000	500	10.880
1'000	1000	13.994

Table 5.3.: Runtimes optimized implementation of Q_QR, fixed row number

Number of rows	Number of columns	execution time in s
1'000	10	0.623
1'000	25	2.263
1'000	50	7.332
1'000	100	26.744
1'000	250	159.204
1'000	500	630.329
1'000	1000	aborted after 30 mins

Table 5.4.: Runtimes brute force implementation of Q_QR, fixed row number

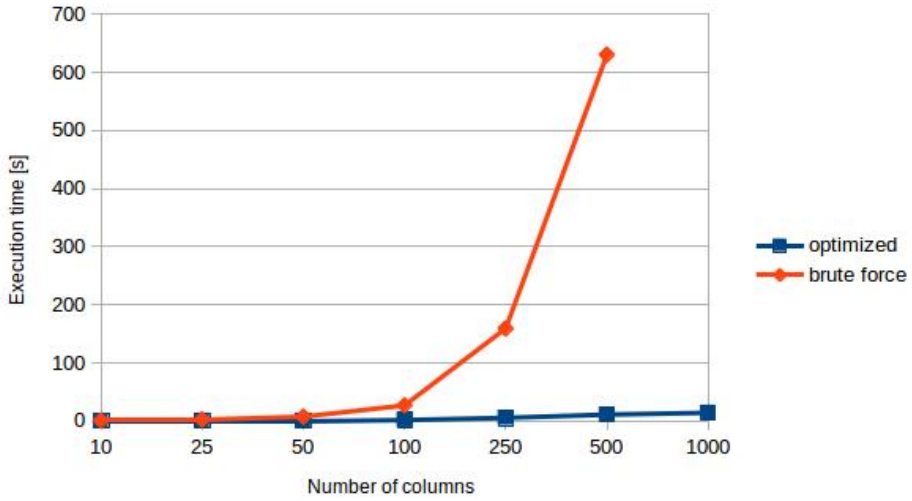

Figure 5.2.: Execution times for Q_QR with a fixed row number of 1000

As can be seen in table 5.3 the execution times for the optimized Q_QR behave approximately with a constant factor times the factor with which the number of rows was mupltiplied(e.g. going from 50 to 500 rows, the execution time is approximately multiplied by a factor of 15). They are generally in a reasonable range for the optimized solution, except for the last value for 1'000 columns, as it is lower than expected. The reason could not be determined though.

However, this is not the case for the brute force solution in table 5.4, where the execution times quickly increase. The last data point is missing for the brute force solution, as it is getting completely unusable. A reason for the fact, that the complexity $\mathcal{O}(m*n^2)$ of is not very evident might be, that the computation of the QRD is computed very quickly for matrices of this size. The other commands (retrieving tuples from the database, allocating memory, etc.) take much longer, also because the QRD is only executed once in the optimized solution. The brute force solution though has shown to react very quickly to bigger table sizes.

6. Summary and Conclusions

The task for this thesis was to integrate the QR decomposition into PostgreSQL using the modified Gram Schmidt algorithm. First, I researched related work, showing how linear algebra and statistical operations are used either in existing database management systems or in conjunction with packages that extend a DBMS's functionality. In a next step, the QR decomposition in general as well as the specific algorithm used for this thesis were introduced, together with an overview in which mathematical context they can be used. In the following chapter, the new operations Q_QR and R_QR were introduced in the context of relational algebra, on which SQL is based. Since there is no guaranteed ordering of rows in SQL, as relational algebra does not define the order of rows, a rowId has been introduced to be able to correctly identify a row and provide a correct result for the QR decomposition. This also implies a fixed schema the implementation relies on.
Another important consideration for the implementation was that the matrix stored in a relation in the database can not simply be accessed with a row and column index, as when computing matrix operations in maths. PostgreSQL is a tuple based DBMS, and as such, a whole row has to be retrieved from the table, even though only a single value might be needed. Therefore, it was decided that the whole table was to be retrieved and stored in a temporary two-dimensional array. This allowed for a for a fast and convenient access via row and column indices while ensuring that the values did not have to be retrieved from the database more often than needed. The presented implementations take these considerations into account.
In a first step an unoptimized brute force implementation was introduced. It has to retrieve the whole input relation for every output row and also has to call the QR decomposition method every time. To improve this implementation, an optimized implementation was introduced. For one, it has to retrieve the input table only once, minimizing time-consuming read operations. Also, the QR decomposition is computed just once.
A further optimization was done based on the examination of the modified Gram Schmidt algorithm. The algorithm has three nested for loops. In the innermost for loops, the matrices are always read column-wise, from top to bottom. This led to the conclusion that the algorithm would execute faster, if the data structure holding the matrices allowed for a sequential read of columns, which was implemented by transposing the array holding the data.

The experiments have shown, that the optimized implementation of Q_QR ran considerably faster than the brute force implementation. Also the execution time behaves relatively linearly when varying the number of rows of a given matrix A in the optimized solution. The brute force solution rather quickly becomes slow and unreliable.

For the experiment with varying column sizes the execution time of the optimized solution mostly takes a constant factor x times the factor with which the column numbers were multiplied longer. In the brute force solution however, the execution time increases almost with the square of the factor, by which the column size is multiplied, and is already unusable for a matrix with the size of (1000x1000), which is no problem for the optimized solution and returns a result quickly.

The optimized implementation presented has shown to be working well, even with quite large input matrices. As such, the target was met and the challenges that occurred were adressed accordingly and a workable implementation has been found. The solution provides a reasonable overhead in the relation itself, as it relies on only one index for every row, which is the least amount of additional indexing possible to ensure correct row ordering. Memory-wise the implementation has its drawbacks. As the input relation as well as the resulting relations are stored in the main memory, it uses a lot of memory if the table sizes are big. This might be a problem on machines with a small main memory, or when computing a lot of concurrent queries calling the function on the same PostgreSQL server. Possibly there might be a solution in the future to save on memory consumption e.g. via storing the resulting matrix Q not in a separate array, but by overwriting array a. Another possibility for the Q_QR operation might be to only store the values of R which are actually necessary for the computation of Q, which could also save memory.

As for including further linear algebra operations into PostgreSQL, it would be crucial to find an intuitive and constructive convention for the schemas used for storing matrices (and vectors) in a table. Indexing is very important for the operations on matrices, as SQL does not define a standard. While it surely is possible to add a lot of frequently used matrix and linear algebra operations, it has to be considered, that new operations add to the complexity of the parser stage and are relatively complex to integrate.

Bibliography

[1] https://www.r-project.org/. Last visit: November 2015.

[2] https://docs.oracle.com/cd/E11882_01/doc.112/e36761/toc.htm. Last visit: November 2015.

[3] http://docs.oracle.com/cd/B19306_01/appdev.102/b14258/u_nla.htm#CIABEFIJ. Last visit: November 2015.

[4] http://www.postgresql.org/docs/9.4/interactive/index.html. Last visit: November 2015.

[5] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[6] Walter Gander. Algorithms for the QR-Decomposition. Technical report, ETH Zurich, April 1980.

[7] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. John Hopkins University Press, 3rd edition, 1996.

[8] Dimitar Misev and Peter Baumann. Homogenizing Data and Metadata Retrieval in Scientific Applications. In *Proceedings of the ACM Eighteenth International Workshop on Data Warehousing and OLAP*, DOLAP '15, pages 25–34. ACM, 2015.

[9] Chitranjan K Singh, Sushma Honnavara Prasad, and Poras T Balsara. VLSI Architecture for Matrix Inversion using Modified Gram-Schmidt based QR Decomposition. In *VLSI Design*, pages 836–841. Citeseer, 2007.

[10] Michael Stonebraker, Paul Brown, Alex Poliakov, and Suchi Raman. The Architecture of SciDB. In *Proceedings of the 23rd International Conference on Scientific and Statistical Database Management*, SSDBM'11, pages 1–16. Springer-Verlag, 2011.

[11] Y. P. Hong, C.-T. Pan. Rank-Revealing QR Factorizations and the Singular Value Decomposition. *Mathematics of Computation*, 58(197):213–232, 1992.

[12] Ying Zhang, Martin Kersten, Milena Ivanova, and Niels Nes. SciQL: Bridging the Gap Between Science and Relational DBMS. In *Proceedings of the 15th Symposium on International Database Engineering & Applications*, IDEAS '11. ACM, 2011.

[13] Ying Zhang, Martin Kersten, and Stefan Manegold. SciQL: Array Data Processing Inside an RDBMS. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1049–1052. ACM, 2013.

Abbreviations

DBMS	database management system
e.g.	example given
i.e.	id est
LS	least squares
MGS	modified Gram Schmidt
QRD	QR decomposition
SVD	singular value decomposition
UDF	user-defined function

A. Installation Guidelines

The installation guideline is provided when downloading the sourcecode from the www.postgresql.org homepage. It is in a file named INSTALL, which can be found in the root folder of the zipped version of the source code (also provided on the CD).

B. Contents of the CD

- Zusfsg.txt Contains the German version of the abstract
- Abstract.txt Contains the English abstract.
- Bachelorarbeit.pdf Contains the complete thesis.
- postgresql-9.4.0.zip Contains the complete source code.